Go 中文文档

wizardforcel

Published
with GitBook



目錄

Go语言文档

Go语言中文小组 翻译整理

2011-03-02

1. 关于本文

本文档是由 Go语言中文小组 根据 golang.org 的文档翻译，最新的翻译文档可以从 http://code.google.com/p/golang-china/获取。

译者列表(如果对署名有争议请联系：chaishushan@gmail.com)：

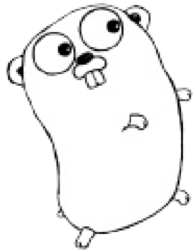
标题	翻译者	校验者
Install Install Go（安装Go环境）	ChaiShushan	
Go Tutorial（Go语言入门教程）	BianJiang && ChaiShushan	ChaiShushan (60%)
Effective Go	BianJiang && ChaiShushan && Fango	
Codelab: How to Write Go Code	GangChen	
Codelab: Writing Web Applications	dworld	
Go For C++ Programmers（C++程序员指南）	BianJiang && ChaiShushan	
Language Specification	Fango	
Memory Model（内存模型）	ChaiShushan	

中英文对照表（待完善）：

英文	中文	解释
Channel	信道	
goroutine	Go程	
slice	切片	

2. Go语言简介

Go语言是由Google开发的一个开源项目，目的之一为了提高开发人员的编程效率。 Go语言语法灵活、简洁、清晰、高效。它对的并发特性可以方便地用于多核处理器 和网络开发，同时灵活新颖的类型系统可以方便地编写模块化的系统。go可以快速编译， 同时具有垃圾内存自动回收功能，并且还支持运行时反射。Go是一个高效、静态类型， 但是又具有解释语言的动态类型特征的系统级语法。



下面是用go编写的"Hello, world"程序：

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, 世界")
}
```

3. 安装go环境

3.1. 简介

Go是一个开源项目，采用BSD授权协议。 该文档介绍如何获取Go源代码，如何编译，以及如何运行Go程序。

目前有两种方式使用Go语言。这里主要讲述如何使用Go专用的gc系列工具 （6g、8g等）。 另一个可选的编译器是 基于gcc后端的 gccgo编译器。关于gccgo的细节清参考安装并使用gccgo编译器 。

Go编译器可以支持三种指令集。不同体系结构生成的代码质量有一些差别：

amd64 (a.k.a. x86-64); 6g,6l,6c,6a

最成熟的实现，编译器在寄存器级别优化，可以生成高质量的目标代码（有时候gccgo可能更优）。

386 (a.k.a. x86 or x86-32); 8g,8l,8c,8a

amd64平台的的完整移植。

arm (a.k.a. ARM); 5g,5l,5c,5a

在完善中。目前只支持生成Linux的二进制文件，浮点支持比较匮乏，并且生成目标代码时还存在bug。还没有完全通过测试集，也没有任何优化。

除了系统级的接口，go需要的运行时环境对各个平台都是一致的。包含 mark-and-sweep 垃圾内存自动回收（更高效的算法实现正在开发中）， 数组、字符串、智能堆栈 以及 goroutine 等。

目前支持以下系统：FreeBSD、Linux、Native Client 和 OS X (a.k.a. Darwin)。 Microsoft Windows 目前正在移植中， 功能还不完整。关于各个系统平台的详细说明，可以参考后面的 [环境变量] 一节。

3.2. 安装C语言工具

Go的工具链采用C语言编写，构建需要安装以下开发工具：

- GCC,
- C语言标准库,
- Bison,
- make,
- awk, 和
- ed （编辑器）.

对于 OS X 系统，以上工具是Xcode的一部分。

对于 Ubuntu/Debian 系统，运行安装命令：

```
sudo apt-get install bison ed gawk gcc libc6-dev make
```

3.3. 安装Mercurial

在进行后面的操作之前需要安装Mercurial版本管理系统（可以输出hg名字检测是否安装）。安装输入以下命令：

```
sudo easy_install mercurial
```

对于 Ubuntu/Debian 系统，easy_install 命令可以用 `apt-get install python-setuptools python-dev build-essential` 安装。

如果上述命令安装失败的话，还可以从 Mercurial Download 下载。

3.4. 获取代码

以下命令会创建一个go目录。切换到相应目录，并且确保当前位置不存在go目录，运行命令：

```
$ hg clone -r release https://go.googlecode.com/hg/ go
```

3.5. 安装Go

编译go环境：

```
$ cd go/src
$ ./all.bash
```

编译完成后，结尾会打印以下信息。

```
--- cd ../test

---
Installed Go for linux/amd64 in /home/you/go.
Installed commands in /home/you/go/bin.
*** You need to add /home/you/go/bin to your $PATH. ***
The compiler is 6g.
```

其中N对于不同的版本会有差异，表示没有通过测试的数目。

3.6. 编写程序

以file.go代码为例，用以下命令编译：

```
$ 6g file.go
```

6g是针对amd64指令的编译器，它的输出文件为 file.6。其中 ‘6’ 表示文件是amd64指令的输出文件。 如果是386和arm 处理器，后缀则为 `8` 和 `5` 。 也就是说，如果你用的是386处理器，那么应该用8g命令编译， 输出的文件为file.8。

然后用以下命令连接：

```
$ 6l file.6
```

运行程序：

```
$ ./6.out
```

一个完整的例子：

```
$ cat >hello.go <<EOF
package main

import "fmt"

func main() {
        fmt.Printf("hello, world\n")
}
EOF
$ 6g hello.go
$ 6l hello.6
$ ./6.out
hello, world
$
```

在连接的时候，没有必要列出hello.6引用的包（这里用到了fmt包）。 连接器（这里是6l）会自动从hello.6文件获取包的引用信息。

如果是编译更复杂的过程，那么可能需要使用Makefile。相关的例子可以 参考 `$GOROOT/src/cmd/godoc/Makefile` 和 `$GOROOT/src/pkg/*/Makefile` 。

3.7. 进一步学习

- 开始阅读 Go语言入门 教程。
- 参考 Wiki Codelab 编写一个web程序。
- 阅读 Effective Go
- 阅读 Go语言文档

3.8. 更新go到新版本

当有新版本发布的时候，会在Go Nuts邮件列表中通知。可以用以下命令获取最新的发布版本：

```
$ cd go/src
$ hg pull
$ hg update release
$ ./all.bash
```

3.9. 社区资源

在 Freenode IRC 上，可能有很多#go-nuts的开发人员和用户，你可以获取即时的帮助。

还可以访问Go语言的官方邮件列表 Go Nuts。

Bug可以在 Go issue tracker 提交。

对于开发Go语言用户，有令一个专门的邮件列表 golang-checkins。 这里讨论的是Go语言仓库代码的变更。

如果是中文用户，请访问：Go语言中文论坛。

3.10. 环境变量

Go编译器需要三个必须的环境变量和一个可选的环境变量。环境变量在.bashrc 或其他配置文件中设置。

$GOROOT

Go安装包的根目录。通常是放在$HOME/go，当然也可以是其他位置。

$GOOS and 和 $GOARCH

这两个环境变量表示目标代码的操作系统和CPU类型。$GOOS选项有linux、freebsd、darwin (Mac OS X 10.5 or 10.6) 和 nacl (Chrome的Native Client接口，还未完成)。$GOARCH的 选项有amd64 (64-bit x86，目前最成熟)、386 (32-bit x86)、 和arm (32-bit ARM，还未完成)。下面是$GOOS和 $GOARCH的可能组合：

```
$GOOS         $GOARCH
darwin        386
darwin        amd64
freebsd       386
freebsd       amd64
linux         386
linux         amd64
linux         arm           incomplete
nacl          386
windows       386           incomplete
```

$GOBIN (optional) （可选）

指明用于存放go的二进制程序目录。如果是没设置$GOBIN环境变量， 则默认是安装在$HOME/bin。如果设置了该变量，需要确保$PATH 变量也包含这个路径，这样编译器可以找到正确的执行文件。

$GOARM (optional, arm, default=6)

ARM处理器（待补充）。

需要说明的是$GOARCH和$GOOS环境变量表示的是目标代码 运行环境，和当前使用的平台是无关的。这个对于交叉编译是很方便的。在.bashrc文件中设置以下环境变量：

```
export GOROOT=$HOME/go
export GOARCH=amd64
export GOOS=linux
export PATH=.:$PATH:$GOBIN
```

检查是否能正常使用:

```
source ~/.bashrc
cd ~
8g -V
```

4. Go语言入门

4.1. 简介

本文是关于Go编程语言的基础教程，主要面向有C/C++基础的读者。它并不是一个语言的完整指南，关于 Go的具体细节请参考 语言规范 一文。在读完这个入门教程后， 深入的华可以继续看 Effective Go ， 这个文档 将涉及到Go语言的更多特性。此外，还有一个《Go语言三日教程》系列讲座： 第一日, 第二日, 第三日。

下面将通过一些小程序来演示go语言的一些关键特性。所有的演示程序都是可以运行的，程序的代码在安装目录的 `/doc/progs/` 子目录中。

文中的代码都会标出在源代码文件中对应的行号。同时为了清晰起见，我们忽略了源代码文件空白行的行号。

4.2. Hello，世界

让我们从经典的"Hello, World"程序开始：

```
05    package main

07    import fmt "fmt"  // Package implementing formatted I/O.

09    func main() {
10        fmt.Printf("Hello, world; or Καλημ?ρα κ?σμε; or こんにち
11    }
```

每个Go源文件开头都有一个 `package` 声明语句，指明源文件所在的包。同时，我们也可以根据具体的需要 来选择导入（ `import` 语句）特定功能的包。在这个例子中，我们通过导入 `fmt` 包来使用我们熟悉的 `printf` 函数。 不过在Go语言中， `Printf` 函数的是大写字母开头，并且以 `fmt` 包名作为前缀： `fmt.Printf` 。

关键字 `func` 用于定义函数。在所有初始化完成后，程序从 `main` 包中的 `main` 函数开始执行。

常量字符串可以包含Unicode字符，采用UTF-8编码。实际上，所有的Go语言源文件都采用UTF-8编码。

代码注释的方式和C++类似：

```
        /* ... */
        // ...
```

稍后，我们还有很多的关于打印的话题。

4.3. 分号（Semicolons）

比较细心的读者可能发现前面的代码中基本没有出现分号 `;` 。其实在go语言中，只有在分隔 for循环的初始化语句时才经常用到；但是代码段末尾的分号一般都是省略的。

当然，你也可以像C或JAVA中那样使用分号。不过在大多数情况下，一个完整语句末尾的分号 都是有go编译器自动添加的——用户不需要输入每个分号。

关于分号的详细描述，可以查看Go语言说明文档。不过在实际写代码时，只需要记得一行末尾的分号 可以省略就可以了（对于一行写多个语句的，可以用分号隔开）。还有一个额外的好处是：在退出 大括号包围的子区域时，分号也是可以省略的。

在一些特殊情况下，甚至可以写出没有任何分号的代码。不过有一个重要的地方：对于"if"等 后面有大括弧的语句，需要将左大括弧放在"if"语句的同一行，如果不这样的话可能出现编译错误。 Go语言强制使用将开始大括弧放在同一行末尾的编码风格。

4.4. 编译

Go是一个编译型的语言。目前有两种编译器，其中"Gccgo"采用GCC作为编译后端。另外还有 根据处理器架构命名的编译器：针对64位x86结构为"6g"，针对32位x86结构的为"8g"等等。 这些go专用的编译器编译很快，但是产生的目标代码效率比gccgo稍差一点。目前（2009年底）， go专用的编译器的运行时系统比"gccgo"要相对健壮一点。

下面看看如何编译并运行程序。先是用针对64位x86结构处理器的“6g”：

```
$ 6g helloworld.go  # 编译; 输出 helloworld.6
$ 6l helloworld.6   # 链接; 输出 6.out
$ 6.out
Hello, world; or Καλημ?ρα κ?σμε; or こんにちは 世界
$
```

如果是用gccgo编译，方法和传统的gcc编译方法类似：

```
$ gccgo helloworld.go
$ a.out
Hello, world; or Καλημ?ρα κ?σμε; or こんにちは 世界
$
```

4.5. Echo

下面的例子是Unix系统中"echo"命令的简单实现：

```
05    package main

07    import (
08        "os"
09        "flag"  // command line option parser
10    )

12    var omitNewline = flag.Bool("n", false, "don't print final

14    const (
15        Space = " "
16        Newline = "\n"
17    )

19    func main() {
20        flag.Parse()   // Scans the arg list and sets up flags
21        var s string = ""
22        for i := 0; i < flag.NArg(); i++ {
23            if i > 0 {
24                s += Space
25            }
26            s += flag.Arg(i)
27        }
28        if !*omitNewline {
29            s += Newline
30        }
31        os.Stdout.WriteString(s)
32    }
```

程序虽然很小，但是包含了go语言的更多特性。在上一个的例子中，我们演示了如何用"func"关键字定义函数。 类似的关键字还有："var"、"const"和"type"等，它们可以用于定义变量、常量和类型等，用法和"import"一致。 我们可以小括弧声明一组类型相同的变量（如7－10和14－17行所示）。当然，也可以分开独立定义：

```
const Space = " "
const Newline = "\n"
```

程序首先导入 os 包，因为后面要用到包中的一个 *os.File 类型的 Stdout 变量。 这里的 import 语句实际上是一个声明，和我们在 hello world 程序中所使用方法一样，包的名字标识符（fmt） 为前缀用于定位包中定位包中的成员，包可以是在当

前目录或标准包目录。在导入包的时候一般会默认选用包本身的 名字（在必要的时候可以将导入的包重新命名）。在“hello world”程序中，我们只是简单的 import "fmt" 。

如果需要，你可以自己重新命名被import的包。但那不是必须的，只在处理包名字冲突的时候会用到。

通过"os.Stdout"，我们可以用包中的"WriteString?"方法来输出字符串。

现在已经导入"flag"包，并且在12行创建了一个全局变量，用于保存echo的"-n"命令行选项。变量 "omitNewline"为一个只想bool变量的bool型指针。

在"main.main"中，我们首先解析命令行参数（20行），然后创建了一个局部字符串变量用于保存要输出的内容。

变量声明语法如下：

```
        var s string = "";
```

这里有一个"var"关键字，后面跟着变量名字和变量的数据类型，再后面可以用“＝”符号来进行赋初值。

简洁是go的一个目标，变量的定义也有更简略的语法。go可以根据初始值来判断变量的类型， 没有必要显式写出数据类型。也可以这样定义变量：

```
        var s = "";
```

还有更短的写法：

s := ""; 操作符":="将在Go中声明同时进行初始化一个变量时会经常使用。下面的代码是在"for"中声明并 初始化变量：

```
22        for i := 0; i < flag.NArg(); i++ {
```

"flag"包会解析命令行参数，并将不是flag选项的参数保存到一个列表中。可以通过flag的参数列表 访问普通的命令行参数。

Go语言的"for"语句和C语言中有几个不同的地方：第一，for是Go中唯一的循环语句，Go中没有while或 do语句；第二，for的条件语句并不需要用小括号包起来，但是循环体却必须要花括弧，这个规则同样适用于 if和switch。后面我们会看到for的一些例子。

在循环体中，通过"+="操作符向字符串"s"添加要命令行参数和空白。在循环结束后，根据命令行是否有"-n"选项， 判断末尾是否要添加换行符。最后输出结果。

值得注意的地方是"main.main"函数并没有返回值（函数被定义为没有返回值的类型）。如果"main.main" 运行到了末尾，就表示“成功”。如果想返回一个出错信息，可用系统调用强制退出：

```
            os.Exit(1)
```

"os"包还包含了其它的许多启动相关的功能，例如"os.Args"是"flag"包的一部分（用来获取命令行输入）。

4.6. 类型简介

Go语言中有一些通用的类型，例如"int"和"float"，它们对应的内存大小和处理器类型相关。同时， 也包含了许多固定大小的类型，例如"int8"和"float64"，还有无符号类型"uint"和"uint32"等。 需要注意的是，即使"int"和"int32"占有同样的内存大小，但并不是同一种数据类型。不过 "byte"和"uint8"对应是相同的数据类型，它们是字符串中字符类型。

go中的字符串是一个内建数据类型。字符串虽然是字符序列，但并不是一个字符数组。可以创建新的 字符串，但是不能改变字符串。不过我们可以通过新的字符串来达到想改变字符串的目的。 下面列举"strings.go"例子说明字符串的常见用法：

```
11          s := "hello"
12          if s[1] != 'e' { os.Exit(1) }
13          s = "good bye"
14          var p *string = &s
15          *p = "ciao"
```

不管如何，试图修改字符串的做法都是被禁止的：

```
        s[0] = 'x';
        (*p)[1] = 'y';
```

Go中的字符串和C++中的"const strings"概念类似，字符串指针则相当于C++中的"const strings" 引用。

是的，它们都是指针，但是Go中用法更简单一些。

数组的声明如下：

```
        var arrayOfInt [10]int;
```

数组和字符串一样也是一个值对象，不过数组的元素是可以修改的。不同于C语言的是："int"类型数组 "arrayOfInt"并不能转化为"int"指针。因为，在Go语言中数组是一个值对象，它在内部保存"int"指针。

数组的大小是数组类型的一部分。我们还可以通过slice（切片）类型的变量来访问数组。 首先，数据元素的类型要和slice（切片）类型相同，然后通过"a: high?"类似的 语法来关联数组的low到heigh-1的子区间元素。Slices和数组的声明语法类似，但是不像数组那样 要指定元素的个数（""和"10?"的区别）；它在内部引用特定的空间，或者其它数组的空间。 如果多个Slices引用同一个数组，则可以共享数组的空间。但是不同数组之间是无法共享内存空间的。

在Go语言中Slices比数组使用的更为普遍，因为它更有弹性，引用的语法也使得它效率很高。 但是，Slices缺少对内存的绝对控制比数组要差一些。例如你只是想要一个可以存放100个元素 的空间，那么你就可以选择数组了。创建数组：

```
[3]int{1,2,3}
```

上面的语句创建一个含有3个元素的int数组。

当需要传递一个数组给函数时，你应该将函数的参数定义为一个Slice。这样，在调用函数的时候， 数组将被自动转换为slice传入。

比如以下函数以slices类型为参数（来自"sum.go"）：

```
09    func sum(a []int) int { // returns an int
10        s := 0
11        for i := 0; i < len(a); i++ {
12            s += a[i]
13        }
14        return s
15    }
```

函数的返回值类型(int)在sum()函数的参数列表后面定义。

为了调用sum函数，我们需要一个slice作为参数。我们先创建一个数组，然后将数组转为slice类型：

```
s := sum([3]int{1,2,3}[:])
```

如果你创建一个初始化的数组，你可以让编译器自动计算数组的元素数目，只要在数组大小中填写"..."就 可以了：

```
s := sum([...]int{1,2,3}[:])
```

是实际编码中，如果不关心内存的具体细节，可以用slice类型（省略数组的大小）来代替数组地址为函数参数：

```
s := sum([]int{1,2,3});
```

还有map类型，可以用以下代码初始化：

```
m := map[string]int{"one":1 , "two":2}
```

用内建的"len()"函数，可以获取map中元素的数目，该函数在前面的"sum"中用到过。"len()"函数 还可以用在strings, arrays, slices, maps, 和 channels 中。

还有另外的"range"语法可以用到strings, arrays, slices, maps, 和 channels 中， 它可以用于"for"循环的迭代。例如以下代码

```
for i := 0; i &lt; len(a); i++ { ... }
```

用"range"语法可以写成：

```
for i, v := range a { ... }
```

这里的"i"对应元素的索引，"v"对应元素的值。关于更多的细节可以参考 Effective Go。

4.7. 申请内存

在Go语言中，大部分的类型都是值变量。例如int或struct(结构体)或array(数组)类型变量， 赋值的时候都是复制整个元素。如果需要为一个值类型的变量分配空间，可以用new()：

```
type T struct { a, b int }
var t *T = new(T);
```

或者更简洁的写法：

```
t := new(T);
```

还有另外一些类型，如：maps, slices 和 channels(见下面)是引用语意（reference semantics）。 如果你一个slice 或 map内的元素，那么其他引用了相同slice 或 map的变量也能看到这个改变。 对于这三类引用类型的变量，需要用另一个内建的make()分配并初始化空间：

```
m := make(map[string]int);
```

上目的代码定义一个新的map并分配了存储空间。如果只是定一个map而不想分配空间的话，可以这样：

```
var m map[string]int;
```

它创建了一个nil(空的)引用并且没有分配存储空间。如果你想用这个map, 你必须使用make来 分配并初始化内存空间或者指向一个已经有存储空间的map。

注意: new(T) 返回的类型是 *T , 而 make(T) 返回的是引用语意的 T 。如果你(错误的)使用 new()` 分配了一个引用对象，你将会得到一个指向 nil 引用的指针。这个相当于声明了一个未初始化引用变量并取得 它的地址。

4.8. 常量

虽然在Go中整数(integer)占用了大量的空间，但是常量类型的整数并没有占用很多空间。 这里没有像0LL 或 0x0UL的常量，取而代之的是使用整数常量作为大型高精度的值。常量只有 在最终被赋值给一个变量的时候才可以会出现溢出的情况：

```
const hardEight = (1 &lt;&lt; 100) &gt;&gt; 97  // legal,
```

具体的语法细节比较琐屑，下面是一些简单的例子：

```
var a uint64 = 0  // a has type uint64, value 0
a := uint64(0)    // equivalent; uses a "conversion"
i := 0x1234       // i gets default type: int
var j int = 1e6   // legal - 1000000 is representable in
x := 1.5          // a float
i3div2 := 3/2     // integer division - result is 1
f3div2 := 3./2\.   // floating point division - result is
```

（强制？）转换只适用于几种简单的情况：转换整数(int)到去其他的精度和大小，整数(int)与 浮点数(float)的转换, 还有其他一些简单情形。在Go语言中，系统不会对两种不同类型变量作 任何隐式的类型转换。此外，由常数初始化的变量需要指定确定的类型和大小。

4.9. I/O包

接下来我们使用open/close/read/write等基本的系统调用实现一个用于文件IO的包。 让我们从文件file.go开始：

```
05    package file

07    import (
08        "os"
09        "syscall"
10    )

12    type File struct {
13        fd   int    // file descriptor number
14        name string // file name at Open time
15    }
```

文件的第一行声明当前代码对应—"file"—包，然后导入os和syscall两个包。 包os封装了不同操作系统底层的实现，例如将文件抽象成相同的类型。我们将在系统接口基础 上封装一个基本的文件IO接口。

另外还有其他一些比较底层的syscall包，它提供一些底层的系统调用(system's calls)。

接下来是一个类型(type)定义：用"type"这个关键字来声明一个类。在这个例子里数据结构(data structure) 名为"File"。为了让这事变的有趣些，我们的File包含了一个这个文件的名字(name)用来描述这个文件。

因为结构体名字"File"的首字母是大写，所以这个类型包(package)可以被外部访问。在GO中访问规则的处理 是非常简单的：如果顶极类型名字首字母(包括：function, method, constant or variable, or of a structure field or method)是大写，那么引用了这个包(package)的使用者就可以访问到它。不然 名称和被命名的东西将只能有package内部看到。这是一个要严格遵循的规则，因为这个访问规则是由编译器(compiler)强制规范的。在GO中，一组公开可见的名称是"exported"。

在这个File例子中，所有的字段(fields)都是小写所以从包外部是不能访问的，不过我们在下面将会一个 一个对外访问的出口(exported) —— 一个以大写字母开头的方法。

首先是一个创建File结构体的函数：

```
17    func newFile(fd int, name string) *File {
18        if fd < 0 {
19            return nil
20        }
21        return &File{fd, name}
22    }
```

这将返回一个指向新File结构体的指针，结构体存有文件描述符和文件名。这段代码使用了GO的复合变量(composite literal)的概念，和创建内建的maps和arrays类型变量一样。要创建在堆（heap-allocated）中创建一个新的 对象，我们可以这样写：

```
            n := new(File);
            n.fd = fd;
            n.name = name;
            return n
```

如果结构比较简单的话，我们可以直接在返回结构体变量地址的时候初始化成员字段，如前面例子的 21行代码所示。

我们可以用前面的函数（newFile）构造一些File类型的变量，返回File：

```
24    var (
25        Stdin  = newFile(0, "/dev/stdin")
26        Stdout = newFile(1, "/dev/stdout")
27        Stderr = newFile(2, "/dev/stderr")
28    )
```

这里的newFile是内部函数，真正包外部可以访问的函数是Open：

```
30    func Open(name string, mode int, perm uint32) (file *File,
31        r, e := syscall.Open(name, mode, perm)
32        if e != 0 {
33            err = os.Errno(e)
34        }
35        return newFile(r, name), err
36    }
```

在这几行里出现了一些新的东西。首先，函数Open返回多个值(multi-value)：一个File指针和一个error(等一下会介绍errors)》我们用括号来表来声明返回多个变量值(multi-value)，语法上它看 起来像第二个参数列表。syscall.Open系统调用同样也是返回多个值multi-value。接着我们能在31行 创建了r和e两个变量用于保存syscall.Open的返回值。函数最终也是返回2个值，分别为File指针和一个error。 如果syscall.Open打开失败，文件描述r将会是个负值，newFile将会返回nil。

关于错误：os包包含了一些常见的错误类型。在用户自己的代码中也尽量使用这些通用的错误。 在Open函数中，我们用os.Error函数将Unix的整数错误代码转换为go语言的错误类型。

现在我们可以创建Files,我们为它定义了一些常用的方法(methods)。要给一个类型定义一个方法(method)， 需要在函数名前增加一个用于访问当前类型的变量。这些是为*File类型创建的一些方法：

```
38    func (file *File) Close() os.Error {
39        if file == nil {
40            return os.EINVAL
41        }
42        e := syscall.Close(file.fd)
43        file.fd = -1 // so it can't be closed again
44        if e != 0 {
45            return os.Errno(e)
46        }
47        return nil
48    }

50    func (file *File) Read(b []byte) (ret int, err os.Error) {
51        if file == nil {
52            return -1, os.EINVAL
53        }
54        r, e := syscall.Read(file.fd, b)
55        if e != 0 {
56            err = os.Errno(e)
57        }
58        return int(r), err
59    }

61    func (file *File) Write(b []byte) (ret int, err os.Error)
62        if file == nil {
63            return -1, os.EINVAL
64        }
65        r, e := syscall.Write(file.fd, b)
66        if e != 0 {
67            err = os.Errno(e)
68        }
69        return int(r), err
70    }

72    func (file *File) String() string {
73        return file.name
74    }
```

这些并没有隐含的this指针（参考C++类），而且类型的方法(methods)也不是定义在struct内部——struct结构只声明数据成员(data members)。事实上，我们可以给任意数据类型定义方法，例如：整数(integer)，数组(array) 等。后面我们会有一个给数组定义方法的例子。

String这个方法之所以会被调用是为了更好的打印信息，我们稍后会详细说明。

方法(methods)使用os.EINVAL来表示(os.Error的版本)Unix错误代码EINVAL。 在os包中针对标准的error变量定义各种错误常量。

现在我们可以使用我们自己创建的包(package)了：

```
05    package main

07    import (
08        "./file"
09        "fmt"
10        "os"
11    )

13    func main() {
14        hello := []byte("hello, world\n")
15        file.Stdout.Write(hello)
16        file, err := file.Open("/does/not/exist",  0,  0)
17        if file == nil {
18            fmt.Printf("can't open file; err=%s\n",  err.String
19            os.Exit(1)
20        }
21    }
```

这个"./"在导入(import)"./file"时告诉编译器(compiler)使用我们自己的package，而不是在 默认的package路径中找。

最后，我们来执行这个程序：

```
    $ 6g file.go                       # compile file package
    $ 6g helloworld3.go                # compile main package
    $ 6l -o helloworld3 helloworld3.6  # link - no need to mentic
    $ helloworld3
    hello, world
    can't open file; err=No such file or directory
    $
```

4.10. Rotting cats

在我们上面创建的file包(package)基础之上，实现一个简单的Unix工具 "cat(1)", "progs/cat.go":

```
05    package main

07    import (
08        "./file"
09        "flag"
10        "fmt"
11        "os"
12    )

14    func cat(f *file.File) {
15        const NBUF = 512
16        var buf [NBUF]byte
17        for {
18            switch nr, er := f.Read(buf[:]); true {
19            case nr < 0:
20                fmt.Fprintf(os.Stderr, "cat: error reading fror
21                os.Exit(1)
22            case nr == 0:  // EOF
23                return
24            case nr > 0:
25                if nw, ew := file.Stdout.Write(buf[0:nr]); nw
26                    fmt.Fprintf(os.Stderr, "cat: error writing
27                }
28            }
29        }
30    }

32    func main() {
33        flag.Parse()   // Scans the arg list and sets up flags
34        if flag.NArg() == 0 {
35            cat(file.Stdin)
36        }
37        for i := 0; i < flag.NArg(); i++ {
38            f, err := file.Open(flag.Arg(i), 0, 0)
39            if f == nil {
40                fmt.Fprintf(os.Stderr, "cat: can't open %s: eri
41                os.Exit(1)
42            }
43            cat(f)
44            f.Close()
45        }
46    }
```

现在应该很容易被理解，但是还有些新的语法"switch". 比如: 包括了"for"循环，"if"和 "switch"初始化的语句。在"switch"语句的18行用了"f.Read()"函数的返回值"nr"和"er"做为 变量(25行中的"if"也采用同样的方法)。这里的"switch"语法和其他语言语法基本相同，每个分支(cases) 从上到下查找是否与相关的表达式相同，分支(case)的表达式不仅仅是常量(constants)或整数(integers), 它可以是你想到的任意类型。

这个"switch"的值永远是"真(true)", 我们会一直执行它, 就像"for"语句，不写值默认是"真"(true). 事实上，"switch"是从"if-else"由来的。在这里我们要说明, "switch"语句中的每个"分支"(case)都 默认隐藏了"break".

在25行中调用"Write()"采用了slicing来取得buffer数据. 在标准的GO中提供了Slices对I/O buffers的操作。

现在让我们做一个"cat"的升级版让"rot13"来处理输入, 就是个简单的字符处理，但是要 采用GO的新特性"接口(interface)"来实现。

这个"cat()"使用了2个子程序"f":"Read()"和"String", 让我们定义这2个接口， 源码参考 "progs/cat_rot13.go"

```
26    type reader interface {
27        Read(b []byte) (ret int, err os.Error)
28        String() string
29    }
```

任何类型的方法都有 reader 这两个方法 —— 也就是说实现了这两个方法， 任何类型的方法都能使用。由于 file.File 实现了 reader 接口，我们就可以让 cat 的子程序访问 reader 从而取代了 *file.File 并且能正常工作，让我们来些第二个类型实现 reader , 一个关注现有的 reader ， 另一个 rot13 只关注数据。我们只是定义了这个类型和 实现了这个方法并没有做其他的内部处理, 我们实现了第二个 reader 接口.

```
31    type rotate13 struct {
32        source    reader
33    }

35    func newRotate13(source reader) *rotate13 {
36        return &rotate13{source}
37    }

39    func (r13 *rotate13) Read(b []byte) (ret int, err os.Error)
40        r, e := r13.source.Read(b)
41        for i := 0; i < r; i++ {
42            b[i] = rot13(b[i])
43        }
44        return r, e
45    }

47    func (r13 *rotate13) String() string {
48        return r13.source.String()
49    }
50    // end of rotate13 implementation
```

(42行的"rot13"函数非常简单，没有必要在这里进行讨论)

为了使用新的特性，我们定义了一个标记(flag):

```
14    var rot13Flag = flag.Bool("rot13", false, "rot13 the input'
```

用它基本上不需要修改"cat()"这个函数：

```
52    func cat(r reader) {
53        const NBUF = 512
54        var buf [NBUF]byte

56        if *rot13Flag {
57            r = newRotate13(r)
58        }
59        for {
60            switch nr, er := r.Read(buf[:]); {
61            case nr < 0:
62                fmt.Fprintf(os.Stderr, "cat: error reading fror
63                os.Exit(1)
64            case nr == 0:  // EOF
65                return
66            case nr > 0:
67                nw, ew := file.Stdout.Write(buf[0:nr])
68                if nw != nr {
69                    fmt.Fprintf(os.Stderr, "cat: error writing
70                }
71            }
72        }
73    }
```

(我们应该对 main 和 cat 单独做些封装，不仅仅是对类型参数的修改，就当是练习) 从56行到 58行: 如果 rot13 标记是真，封装的 reader 就会接受数据并传给 rotate13 并处理. 注意: 这个接口的值是变量，不是指针，这个参数是 reader 类型，不是 *reader , 尽管后面转换为 指向结构体的指针。

这里是执行结果:

```
% echo abcdefghijklmnopqrstuvwxyz | ./cat
abcdefghijklmnopqrstuvwxyz
% echo abcdefghijklmnopqrstuvwxyz | ./cat --rot13
nopqrstuvwxyzabcdefghijklm
%
```

也许你会说使用注入依赖(dependency injection)能轻松的让接口以一个文件描述符执行。

接口(interfaces)是Go的一个特性，一个接口是由类型实现的，接口就是声明该类型的所有方法。 也就是说一个类型可以实现多个不同的接口, 没有任何类型的限制, 就像我们的例子"rot13". "file.File"这个类型实现了"reader", 它也能实现"writer", 或通过其他的方法来实现这个接口。 参考空接口(empty interface)

```
type Empty interface {}
```

任何类型都默认实现了空接口，我们可以用空接口来保存任意类型。

4.11. Sorting

接口(interfaces)提供了一个简单形式的多态(polymorphism). 他们把对象的定义和如何实现的分开处理，允许相同的接口可以有不能的实现方法。

参考这个简单的排序算法(sort algorithm)"progs/sort.go"

```
13    func Sort(data Interface) {
14        for i := 1; i < data.Len(); i++ {
15            for j := i; j > 0 && data.Less(j, j-1); j-- {
16                data.Swap(j, j-1)
17            }
18        }
19    }
```

我们要封装这个排序(sort)的接口(interface)仅需要三个方法。

```
07    type Interface interface {
08        Len() int
09        Less(i, j int) bool
10        Swap(i, j int)
11    }
```

我们可以用任何类型的"Sort"去实现"Len", "Less" 和 "Swap". 这个"sort"包里面 包含一些方法(methods). 下面是整型数组的代码:

```
33    type IntArray []int

35    func (p IntArray) Len() int            { return len(p) }
36    func (p IntArray) Less(i, j int) bool  { return p[i] < p[j]
37    func (p IntArray) Swap(i, j int)       { p[i], p[j] = p[j],
```

你看到的是一个没有任何类型的"结构体"(non-struct type). 在你的包里面你可以定义 任何你想定义的类型.

现在用"progs/sortmain.go"程序进行测试，用"sort"包里面的排序函数进行排序。

```
12    func ints() {
13        data := []int{74, 59, 238, -784, 9845, 959, 905, 0, 0,
14        a := sort.IntArray(data)
15        sort.Sort(a)
16        if !sort.IsSorted(a) {
17            panic("fail")
18        }
19    }
```

如果我们为sort提供一个新类型，我们就需要为这个类型实现三个方法，如下:

```
30    type day struct {
31        num        int
32        shortName  string
33        longName   string
34    }

36    type dayArray struct {
37        data []*day
38    }

40    func (p *dayArray) Len() int            { return len(p.data
41    func (p *dayArray) Less(i, j int) bool  { return p.data[i]
42    func (p *dayArray) Swap(i, j int)       { p.data[i], p.data
```

4.12. 打印输出

前面例子中涉及到的打印都比较简单。在这一节中，我们将要讨论Go语言格式化输出的功能。

我们已经用过"fmt"包中的"Printf"和"Fprintf"等输出函数。"fmt"包中的"Printf"函数的完整说明如下：

```
        Printf(format string, v ...) (n int, errno os.Error)
```

其中"..."表示数目可变参数，和C语言中"stdarg.h"中的宏类似。不过Go中，可变参数是通道 一个空接口（"interface {}"）和反射（reflection）库实现的。反射特性可以帮助"Printf" 函数很好的获取参数的详细特征。

在C语言中，printf函数的要格式化的参数类型必须和格式化字符串中的标志一致。不过在Go语言中， 这些细节都被简化了。我们不再需要"%llud"之类的标志，只用"%d"表示要输出一个整数。至于对应 参数的实际类型，"Printf"可以通过反射获取。例如：

```
  10        var u64 uint64 = 1<<64-1
  11        fmt.Printf("%d %d\n", u64, int64(u64))
```

输出

```
        18446744073709551615 -1
```

最简单的方法是用"%v"标志，它可以以适当的格式输出任意的类型（包括数组和结构）。下面的程序，

```
  14        type T struct {
  15            a int
  16            b string
  17        }
  18        t := T{77, "Sunset Strip"}
  19        a := []int{1, 2, 3, 4}
  20        fmt.Printf("%v %v %v\n", u64, t, a)
```

将输出：

```
        18446744073709551615 {77 Sunset Strip} [1 2 3 4]
```

如果是使用"Print"或"Println"函数的话，甚至不需要格式化字符串。这些函数会针对数据类型 自动作转换。"Print"函数默认将每个参数以"%v"格式输出，"Println"函数则是在"Print"函数 的输出基础上增加一个换行。一下两种输出方式和前面的输出结果是一致的。

```
21        fmt.Print(u64, " ", t, " ", a, "\n")
22        fmt.Println(u64, t, a)
```

如果要用"Printf"或"Print"函数输出似有的结构类型，之需要为该结构实现一个"String()"方法， 返回相应的字符串就可以了。打印函数会先检测该类型是否实现了"String()"方法，如果实现了则以 该方法返回字符串作为输出。下面是一个简单的例子。

```
09    type testType struct {
10        a int
11        b string
12    }

14    func (t *testType) String() string {
15        return fmt.Sprint(t.a) + " " + t.b
16    }

18    func main() {
19        t := &testType{77, "Sunset Strip"}
20        fmt.Println(t)
21    }
```

因为 *testType 类型有 String() 方法，因此格式化函数用它作为输出结果：

```
        77 Sunset Strip
```

前面的例子中，"String()"方法用到了"Sprint"（从字面意思可以猜测函数将返回一个字符串） 作为格式化的基础函数。在Go中，我们可以递归使用"fmt"库中的函数来为格式化服务。

"Printf"函数的另一种输出是"%T"格式，它输出的内容更加详细，可以作为调试信息用。

自己实现一个功能完备，可以输出各种格式和精度的函数是可能的。不过这不是该教程的重点，大家 可以把它当作一个课后练习。

读者可能有疑问，"Printf"函数是如何知道变量是否有"String()"函数实现的。实际上，我们 需要先将变量转换为Stringer接口类型，如果转换成功则表示有"String()"方法。下面是一个 演示的例子：

```
type Stringer interface {
        String() string
}

s, ok := v.(Stringer);  // Test whether v implements "Str
if ok {
        result = s.String()
} else {
        result = defaultOutput(v)
}
```

这里用到了类型断言("v.(Stringer)")，用来判断变量"v"是否可以满足"Stringer"接口。 如果满足，"s"将对应转换后的Stringer接口类型并且"ok"被设置为"true"。然后我们通过"s"， 以Stringer接口的方式调用String()函数。如果不满足该接口特征，"ok"将被设置为false。

"Stringer"接口的命名通常是在接口方法的名字后面加e?r后缀，这里是"String+er"。

Go中的打印函数，除了"Printf"和"Sprintf"等之外，还有一个"Fprintf"函数。不过"Fprintf"函数和 的第一个参数并不是一个文件，而是一个在"io"库中定义的接口类型：

```
type Writer interface {
        Write(p []byte) (n int, err os.Error);
}
```

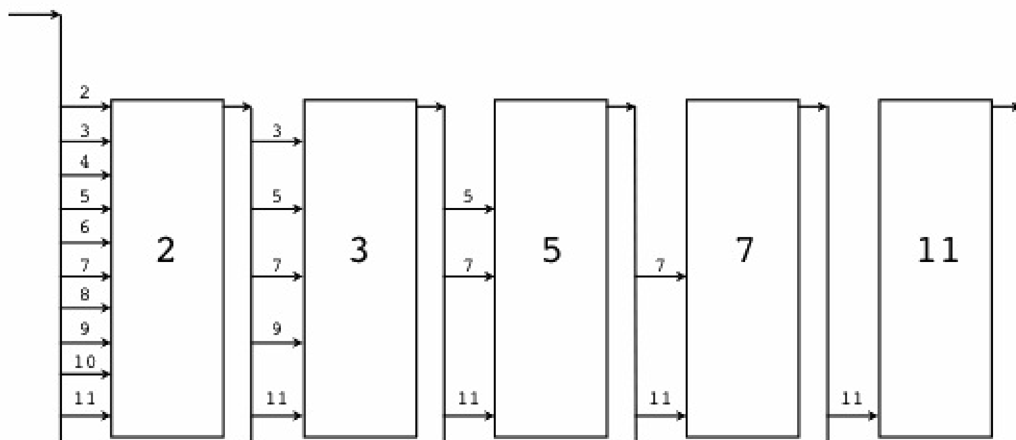
这里的接口也是采用类似的命名习惯，类型的接口还有"io.Reader"和"io.ReadWriter?"等。 在调用"Fprintf"函数时，可以用实现了"Write"方法的任意类型变量作为参数，例如文件、网络、管道等等。

4.13. 生成素数

这里我们要给出一个并行处理程序及之间的通信。这是一个非常大的课题，我们这里只是给出一些要点。

素数筛选是一个比较经典的问题（这里侧重于Eratosthenes素数筛选算法的并行特征）。它以全部的 自然后为筛选对象。首选从第一个素数2开始，后续数列中是已经素数倍数的数去掉。每次筛选可以得到 一个新的素数，然后将新的素数加入筛选器，继续筛选后面的自然数列（这里要参考算法的描述调整）。

这里是算法工作的原理图。每个框对应一个素数筛选器，并且将剩下的数列传给下一个素数筛进行筛选。



为了产生整数序列，我们使用管道。管道可以用于连接两个并行的处理单。在Go语言中， 管道由运行时库管理，可以用"make"来创建新的管道。

这是"progs/sieve.go"程序的第一个函数：

```
09    // Send the sequence 2, 3, 4, ... to channel 'ch'.
10    func generate(ch chan int) {
11        for i := 2; ; i++ {
12            ch <- i  // Send 'i' to channel 'ch'.
13        }
14    }
```

函数"generate"用于生成2, 3, 4, 5, ...自然数序列，然后依次发送到管道。 这里用到了二元操作符"<-"， 它用于向管道发送数据。当管道没有接受者的时候 会阻塞，直到有接收者从管道接受数据为止。

过滤器函数有三个参数：输入输出管道和用于过滤的素数。当输入管道读出来的数不能被 过滤素数整除时，则将当前整数发送到输出管道。这里用到了"<-"操作符，它用于从 管道读取数据。

```
16    // Copy the values from channel 'in' to channel 'out',
17    // removing those divisible by 'prime'.
18    func filter(in, out chan int, prime int) {
19        for {
20            i := <-in  // Receive value of new variable 'i' fro
21            if i % prime != 0 {
22                out <- i  // Send 'i' to channel 'out'.
23            }
24        }
25    }
```

整数生成器generator函数和过滤器filters是并行执行的。Go语言有自己的并发 程序设计模型，这个和传统的进程/线程/轻量线程类似。为了区别，我们把Go语言 中的并行程序称为goroutines。如果一个函数要以goroutines方式并行执行， 只要用"go"关键字作为函数调用的前缀即可。goroutines和它的启动线程并行执行， 但是共享一个地址空间。例如，以goroutines方式执行前面的sum函数：

```
go sum(hugeArray); // calculate sum in the background
```

如果想知道计算什么时候结束，可以让sum用管道把结果返回：

```
ch := make(chan int);
go sum(hugeArray, ch);
// ... do something else for a while
result := &lt;-ch;  // wait for, and retrieve, result
```

再回到我们的素数筛选程序。下面程序演示如何将不同的素数筛链接在一起：

```
28    func main() {
29        ch := make(chan int)  // Create a new channel.
30        go generate(ch)  // Start generate() as a goroutine.
31        for {
32            prime := <-ch
33            fmt.Println(prime)
34            ch1 := make(chan int)
35            go filter(ch, ch1, prime)
36            ch = ch1
37        }
38    }
```

29行先调用"generate"函数，用于产生最原始的自然数序列（从2开始）。然后 从输出管道读取的第一个数为新的素数，并以这个新的素数生成一个新的过滤器。 然后将新创建的过滤器添加到前一个过滤器后面，新过滤器的输出作为新的输出 管道。

sieve程序还可以写的更简洁一点。这里是"generate"的改进，代码在"progs/sieve1.go"中：

```
10    func generate() chan int {
11        ch := make(chan int)
12        go func(){
13            for i := 2; ; i++ {
14                ch <- i
15            }
16        }()
17        return ch
18    }
```

新完善的generate函数在内部进行必须的初始化操作。它创建输出管道，然后 启动goroutine用于产生整数序列，最后返回输出管道。它类似于一个并发程序 的工厂函数，完成后返回一个用于链接的管道。

第12-16行用go关键字启动一个匿名函数。需要注意的是，generate函数的"ch" 变量对于匿名函数是可见，并且"ch"变量在generate函数返回后依然存在（因为 匿名的goroutine还在运行）。

这里我们采用过滤器"filter"来筛选后面的素数：

```
21    func filter(in chan int, prime int) chan int {
22        out := make(chan int)
23        go func() {
24            for {
25                if i := <-in; i % prime != 0 {
26                    out <- i
27                }
28            }
29        }()
30        return out
31    }
```

函数"sieve"对应处理的一个主循环，它只是依次将数列交给后面的素数筛选器进行筛选。 如果遇到新的素数，再输出素数后以该素数创建信的筛选器。

```
33    func sieve() chan int {
34        out := make(chan int)
35        go func() {
36            ch := generate()
37            for {
38                prime := <-ch
39                out <- prime
40                ch = filter(ch, prime)
41            }
42        }()
43        return out
44    }
```

主函数入口启动素数生成服务器，然后打印从管道输出的素数：

```
46    func main() {
47        primes := sieve()
48        for {
49            fmt.Println(<-primes)
50        }
51    }
```

4.14. Multiplexing

基于管道，我们可以很容易实现一个支持多路客户端的服务器程序。采用的技巧是将每个客户端私有的通信管道 作为消息的一部分发送给服务器，然后服务器通过这些管道和客户端独立通信。现实中的服务器实现都很复杂， 我们这里只给出一个服务器的简单实现来展现前面描述的技巧。首先定义一个"request"类型，里面包含一个 客户端的通信管道。

```
09    type request struct {
10        a, b    int
11        replyc  chan int
12    }
```

服务器对客户端发送过来的两个整数进行运算。下面是具体的函数，函数在运算完之后将结构通过结构中的 管道返回给客户端。

```
14    type binOp func(a, b int) int

16    func run(op binOp, req *request) {
17        reply := op(req.a, req.b)
18        req.replyc <- reply
19    }
```

第14行现定义一个"binOp"函数类型，用于对两个整数进行运算。

服务器routine线程是一个无限循环，它接受客户端请求。然后为每个客户端启动一个独立的routine线程， 用于处理客户数据（不会被某个客户端阻塞）。

```
21    func server(op binOp, service chan *request) {
22        for {
23            req := <-service
24            go run(op, req)  // don't wait for it
25        }
26    }
```

启动服务器的方法也是一个类似的routine线程，然后返回服务器的请求管道。

```
28    func startServer(op binOp) chan *request {
29        req := make(chan *request)
30        go server(op, req)
31        return req
32    }
```

这里是一个简单的测试。首先启动服务器，处理函数为计算两个整数的和。接着向服务器发送"N"个请求（无阻塞）。 当所有请求都发送完了之后，再进行验证返回结果。

```
34    func main() {
35        adder := startServer(func(a, b int) int { return a + b
36        const N = 100
37        var reqs [N]request
38        for i := 0; i < N; i++ {
39            req := &reqs[i]
40            req.a = i
41            req.b = i + N
42            req.replyc = make(chan int)
43            adder <- req
44        }
45        for i := N-1; i >= 0; i-- {   // doesn't matter what or
46            if <-reqs[i].replyc != N + 2*i {
47                fmt.Println("fail at", i)
48            }
49        }
50        fmt.Println("done")
51    }
```

前面的服务器程序有个小问题：当main函数退出之后，服务器没有关闭，而且可能有一些客户端被阻塞在 管道通信中。为了处理这个问题，我们可给服务器增加一个控制管道，用于退出服务器。

```
32    func startServer(op binOp) (service chan *request, quit cha
33        service = make(chan *request)
34        quit = make(chan bool)
35        go server(op, service, quit)
36        return service, quit
37    }
```

首先给"server"函数增加一个控制管道参数，然后这样使用：

```
21    func server(op binOp, service chan *request, quit chan bool
22        for {
23            select {
24            case req := <-service:
25                go run(op, req)  // don't wait for it
26            case <-quit:
27                return
28            }
29        }
30    }
```

在服务器函数中，"select"操作服用于从多个通讯管道中选择一个就绪的管道。如果所有的管道都没有数据， 那么将等待知道有任意一个管道有数据。如果有多个管道就绪，则随即选择一个。服务器处理客户端请求，如果 有退出消息则退出。

最后是在main函数中保存"quit"管道，然后在退出的时候向服务线程发送停止命令。

```
40        adder, quit := startServer(func(a, b int) int { return
...

55        quit <- true
```

当然，Go语言及并行编程要讨论的问题很多。这个入门只是给出一些简单的例子。

5. Effective Go

5.1. 简介

Go是一个新的语言。虽然它从其他语言中借鉴了一些特性，但是Go语言的编程方式和其他是 有本质却别的。如果只是简单的将C++或Java等代码翻译为Go代码是不可能得到最优的Go代码的。 java程序员用java的思维方式编程，并不是Go的思维方式。如果采用go的思维方式，一个问题 可能有完全不同的解决方法。因此，如果要真正的用好Go语言，理解它的语言特性和设计思想是 很重要的。另外，还要知道Go语言的变成风格，例如命名方式、格式化、程序结构等等，采用通用 的方式也便于和其他的Go程序员交流。

该文档对于如何编写清晰优雅的Go程序给出一些建议。它是Go语法说明 和Go语言入门教程的补充。

5.1.1. 例子

Go源代码不仅包含了核心库的实现，还有很多如何使用语言的例子。 如果在使用go的过程中遇到问题，或者想了解某些库的内部工作机制，可以直接参考源代码找到答案。

5.2. 格式化

格式化是一个最有争议的问题。虽然人可以适应各种不同的风格，不过如果大家都遵循一个默认统一的 风格是最理想的。当然，这也是一个仁者见仁、智者见智的问题，不可能有一个终极的理想答案。

对于Go语言，我们采用不同的处理方法：让机器处理绝大部分的格式化工作。工具程序 gofmt可以根据需要将Go代码格式自动格式化为统一的风格。如果你想 了解格式化后代码的缩进方式，你可以直接运行gofmt，然后查看输出结果。

下面是一个例子，我们没有必要花时间手工调整类型中成员注释的对齐方式。Gofmt 可以自动将注释对齐。下面是结果的定义：

```
type T struct {
    name string // name of the object
    value int // its value
}
```

gofmt处理后的结果:

```
type T struct {
    name    string // name of the object
    value   int    // its value
}
```

Go语言库中的所有代码都是用gofmt工具格式化的。

格式化的一些细节：

缩进

我们使用tab缩进，gofmt也是默认用tab缩进。当然，也可以指定空白缩进。

行的长度

Go语言代码每行长度没有限制。不用担心一行的代码太长超出显式范围，gofmt会自动 处理太长的行。

小括号

Go语言很少使用括弧：对于控制结构(if,for,switch) 括弧也不是必须的。而且Go中表达式中运算符的优先级比较简洁，例如下面代码：

```
x&lt;&lt;8 + y&lt;&lt;16
```

意思是x和y移位后相加。

5.3. 注释

Go支持C语言风格的 `/* */` 块注释，也支持C＋＋风格的 `//` 行注释。 当然，行注释更通用，块注释主要用于针对包的详细说明或者屏蔽大块的代码。

程序 - 也是网页服务器 - godoc 处理 Go 的源代码，从中提取包的文档。顶层声明前的注解，如无空行相隔，和声明一起提取作为条目的解释文字。这些注解的性质和风格决定着 godoc 产生的文档的质量。

每个包都应有一个包注解，即 package 前的块注解。对多个文件的包，包注解只需出现在一个文件中，随便哪个。包注解应该介绍此包，并作为一个整体提供此包的对应信息。它首先出现在 godoc 页面，来安排好后续的详细文档

```
/*
    The regexp package implements a simple library for
    regular expressions.

    The syntax of the regular expressions accepted is:

    regexp:
        concatenation { '|' concatenation }
    concatenation:
        { closure }
    closure:
        term [ '*' | '+' | '?' ]
    term:
        '^'
        '$'
        '.'
        character
        '[' [ '^' ] character-ranges ']'
        '(' regexp ')'
*/
package regexp
```

包如果简单，注释可以简短。

```
// The path package implements utility routines for
// manipulating slash-separated filename paths.
```

注解不需多余排版如星星横幅等。生成的结果呈现时可能不是等宽字体，所以不要靠空格对齐， godoc，类似 gofmt 照管这些。最后，注解是不加解释的文本，HTML和其他例如 *this* 会原样照搬，所以应 避免使用。

在包里，紧跟顶层声明前的注解作为此声明的文注解，程序中每个导出（大写）的名字都应该有文注解。

文注解最好是完整的句子。首句应该以声明的名字开始的一句话的总结。

```
// Compile parses a regular expression and returns, if successful
// object that can be used to match against text.
func Compile(str string) (regexp *Regexp, error os.Error) {
```

Go 的声明句法允许编组。单一的文注解可以引出一组相联的常量或变量。因为整组声明一起展现，注解可以很粗略：

```
// Error codes returned by failures to parse an expression.
var (
    ErrInternal      = os.NewError("internal error")
    ErrUnmatchedLpar = os.NewError("unmatched '('")
    ErrUnmatchedRpar = os.NewError("unmatched ')'")
    ...
)
```

对于私有名称，编组也可以指出它们之间的联系，例如一系列的变量由一个互斥保护。

```
var (
    countLock   sync.Mutex
    inputCount  uint32
    outputCount uint32
    errorCount  uint32
)
```

5.4. 命名

名称在 Go 里和在其它语言里一样重要。某种情况下它们甚至有语义效果：例如，一个名称能否在包外可见取决于它的第一个字母是否大写。所以值得花点时间探讨下 Go 程序的命名约定：

5.4.1. 包的命名

当包引入时，包名成为其内容的引导符。

```
import "bytes"
```

后， 导入者可以讲 bytes.Buffer。更有用的是每个包的用户都能使用相同的名称指出它的内容，亦即包应有个好名称：短，精，好记。习惯上包名是小写的单字的名称；应无必要用下划线或大小混写。简错不纠，因为你的包的每个用户都要敲这个名字。还有不要无谓烦扰撞名。包名只是引入时的默认名；它不需在所有源码中都唯一，如出现少见的撞名，导入者可以给出不同的名字局部使用。无论如何，撞名很少见，因为 import 用的文件名只决定使用那个包。

另一个习惯是包名是源目录的基名；src/pkg/container/vector 里的包引入为 “container/vector” 但包名是 vector，不是 container_vector 也不是 containerVector。

导入者使用包名引导其内容 （import . 的记法主要特意用在测试或其它不寻常的场合），所以包的导出的名称可据此避免结结巴巴。例如，bu?o 包的 buffered reader 叫 Reader，不叫 BufReader，因为用户看到的是 bu?o.Reader 这个清楚简短的名称。再有，因为导入项总是给出其包名，bu?o.Reader 不会和 io.Reader 撞名。类似的，用来生成 ring.Ring 的函数 — 即 Go 的架构函数 — 通常会被称为 NewRing，但因为 Ring 是此包唯一的导出类型，并且既然包名叫ring，它就叫 New。此包的客户看到的是 ring.New。使用包结构帮你来选个好名。

Another short example is once.Do; once.Do(setup) reads well and would not be improved by writing once.DoOrWaitUntilDone(setup). Long names don't automatically make things more readable. If the name represents something intricate or subtle, it's usually better to write a helpful doc comment than to attempt to put all the information into the name.

5.4.2. 接口的命名

习惯上，单一成员的界面的名称是其成员名加 -er：Reader, Writer,Formatter 等。

存在这样的一些名称，尊重它们和它们所指的函数会工作的更好。Read, Write, Close, Flush, String 等保有正统的签名和意义。为了避免混淆， 除非有同样的签名和意义，不要给你的方法这些名字。同理，如果你的方法实现了和这些著名方法同样的意图，给它同样的名称和签名；叫你的字符转换器 String 而不是 ToString。

5.4.3. 大小写混写

最后，Go 习惯使用 MixedCaps 和 mixedCaps，而不是下划线来写多字的名称。

5.5. 分号

Go语言与C一样都是采用分号来结束一条语句，不一样的是，并不是所有的源码 都要使用分号。Go是采用语法解析器自动在每行末增加分号，所有你在写代码的 时候可以把分号缩略.

这个规则是: 如果一个标记(token)的前一行是标识符(identifier)(就像"int"或"float64"), 比如: 数字，一个字符串或一个标记.

```
break continue fallthrough return ++ -- ) }
```

那么语法解析器就会在标记的后面插入分号，也就是说"在标记的后面是个换行，这说明可能是语句的结束，就增加一个分号"。

在右括号之前可以省略分号，比如:

```
    go func() { for { dst <- <-src } }()
```

不需要分号。在Go编程中只有几个地方需要增加分号， 比如: for循环 为了把初始化，条件和遍历元素分开。还有在一行中有多条语句，也需要增加分号。

需要注意的是，你不能把控制语句(if, for, switch, or select)左大括号单独方在一行， 如果你这样作了在大括号之前将要插入一个分号，可能会造成不必要的麻烦，要写成:

```
if i < f() {
    g()
}
```

不要写成

```
if i < f()  // wrong!
{           // wrong!
    g()
}
```

5.6. 控制流

Go语言的控制结构与C的基本相同但是有些地方还是不同。Go中没有do， while这样的循环，for与switch 也非常的灵活。if和switch可以有一个初始化语句 就像for一样。还增加了一个type switch(类型选择)和多通道复用(multiway communications multiplexer)的select. 语法有一点点区别，圆括号大部分是 不需要的但是大括号必须始终括号分隔.

5.6.1. If

Go中简单的if实例:

```
if x > 0 {
    return y
}
```

建议在写if语句的时候采用多行。这是一种非常好的编程风格， 特别是在控制结构体里面有return或break的时候.

if和switch允许初始化声明，就可以使用本地变量(local variable).

```
if err := file.Chmod(0664); err != nil {
    log.Stderr(err)
    return err
}
```

在Go的库文件中，你可能经常看到if语句不进入下一条语句是因为函数在break,continue,goto或reurn结束, else可以省略。

```
f, err := os.Open(name, os.O_RDONLY, 0)
if err != nil {
    return err
}
codeUsing(f)
```

一般的代码都会考虑错误的处理，如果没有出错的情况就继续运行但是在出错的时候 函数就会返回，所以在这里不需要else语句。

```
f, err := os.Open(name, os.O_RDONLY, 0)
if err != nil {
    return err
}
d, err := f.Stat()
if err != nil {
    return err
}
codeUsing(f, d)
```

5.6.2. For

Go中的for循环与C相似，但是也有不同的地方。Go只有for和 while, 没有do-while语句。这里有三种方式，只有一种方式 使用了分号。

```
// Like a C for
for init; condition; post { }

// Like a C while
for condition { }

// Like a C for(;;)
for { }
```

短声明使得在循环里声明下标变量很容易。

```
sum := 0
for i := 0; i < 10; i++ {
    sum += i
}
```

如果你要遍历一个array, slice, string or map or 从通道(channel)读数 range将是你最好的选择。

```
var m map[string]int
sum := 0
for _, value := range m {  // key is unused
    sum += value
}
```

对于字符串，range能为你更好的工作，比如解析UTF-8并单独输出Unicode字符.

```
for pos, char := range "日本語" {
    fmt.Printf("character %c starts at byte position %d\n", char,
}
```

打印出：

```
character 日 starts at byte position 0
character 本 starts at byte position 3
character 語 starts at byte position 6
```

最后，Go中没有逗号运算符(comma operator)和++与--运算， 如果你想执行多个变量在for语句中，你可以使用并行参数(parallel assignment).

```
// Reverse a
for i, j := 0, len(a)-1; i < j; i, j = i+1, j-1 {
    a[i], a[j] = a[j], a[i]
}
```

5.6.3. Switch

Go中的switch要比C更全面，C的表达式仅仅只有常数或整数。 每个分支(cases)从上到下进行匹配取值, 如果switch没有表达式 那么switches是真。所有才有可能使用switch替换

```
func unhex(c byte) byte {
    switch {
    case '0' <= c && c <= '9':
        return c - '0'
    case 'a' <= c && c <= 'f':
        return c - 'a' + 10
    case 'A' <= c && c <= 'F':
        return c - 'A' + 10
    }
    return 0
}
```

没有自动掉到下一分支，但分支可以是逗号分隔的列表：

```
func shouldEscape(c byte) bool {
    switch c {
    case ' ', '?', '&', '=', '#', '+', '%':
        return true
    }
    return false
}
```

这个操作数组的程序和上面的相似是通过两个switch语句。

```
// Compare returns an integer comparing the two byte arrays
// lexicographically.
// The result will be 0 if a == b, -1 if a < b, and +1 if a > b
func Compare(a, b []byte) int {
    for i := 0; i < len(a) && i < len(b); i++ {
        switch {
        case a[i] > b[i]:
            return 1
        case a[i] < b[i]:
            return -1
        }
    }
    switch {
    case len(a) < len(b):
        return -1
    case len(a) > len(b):
        return 1
    }
    return 0
}
```

switch可以动态的取得接口变量的数据类型，比如: type switch就是 用关键字type插入到接口类型后面的括号来判断类型， 如果switch 在表达式中声明了一个变量，在分支上就有相应的类型。

```
switch t := interfaceValue.(type) {
default:
    fmt.Printf("unexpected type %T", t)  // %T prints type
case bool:
    fmt.Printf("boolean %t\n", t)
case int:
    fmt.Printf("integer %d\n", t)
case *bool:
    fmt.Printf("pointer to boolean %t\n", *t)
case *int:
    fmt.Printf("pointer to integer %d\n", *t)
}
```

5.7. 函数

5.7.1. 多值返回

Go语言中函数和方法方法的一个有意思的特性是它们可以同时返回多个值。它可以比C语言 更简洁的处理多个返回值的情况：例如在修改一个参数的同时获取错误返回值 （-1或EOF）。

在传统的C语言中，如果写数据失败的话，会在另外一个地方保存错误标志，而且错误标志很容易被 其他函数产生的错误覆盖。在Go语言中，则可以在返回成功写入的数据数目的同时，也可以返回有意义 的错误信息："您已经写了一些数据，但不是全部，因为设备在阻塞填充中"。对于os 包中的*File.Write函数，说明如下：

```
func (file *File) Write(b []byte) (n int, err Error)
```

在函数的文档中有函数返回值的描述：返回成功写入的数据长度，如果n != len(b)，则同时返回一个非non-nil的错误信息。 这是Go语言中，处理错误的常见方式。在后面的"错误处理"一节，会有更多的描述。

多个返回值还可以用于模拟C语言中通过指针的方式遍历。下面的函数是从一个int数组中获取 一个数据，然后移动到下一个位置。

```
func nextInt(b []byte, i int) (int, int) {
    for ; i < len(b) && !isDigit(b[i]); i++ {
    }
    x := 0
    for ; i < len(b) && isDigit(b[i]); i++ {
        x = x*10 + int(b[i])-'0'
    }
    return x, i
}
```

你还可以用这个方法来打印一个数组：

```
    for i := 0; i < len(a); {
        x, i = nextInt(a, i)
        fmt.Println(x)
    }
```

5.7.2. 命名的结果参数

Go语言中，我们还可以给函数或方法的返回值命名，就像函数的输入参数那样。如果我们命名了返回值， 那么它们将在函数开始的时候被初始化为空。然后，在执行不带参数的return语句时， 命名的返回值变量将被用于返回。

返回值命名并不强制使用，但是有时我们给名返回值命令可以产生更清晰的代码，同时它也可以用于文档。 例如，我们把nextInt的返回值命名：

```
func nextInt(b []byte, pos int) (value, nextPos int) {
```

命名后，返回值会自动初始化，而且不需要在return中显式写出返回参数。下面的io.ReadFull 函数是另一个类似的例子：

```
func ReadFull(r Reader, buf []byte) (n int, err os.Error) {
    for len(buf) > 0 && err == nil {
        var nr int
        nr, err = r.Read(buf)
        n += nr
        buf = buf[nr:len(buf)]
    }
    return
}
```

5.7.3. Defer

Go 的 defer 语句安排一个函数调用（被 defer 的函数）延迟发生在执行 defer 的函数刚要返回之前。当函数无论怎样返回，某资源必须释放时，可用这种与众不同、但有效的处理方式。传统的例子包括解锁互斥或关闭文件。

```
// Contents returns the file's contents as a string.
func Contents(filename string) (string, os.Error) {
    f, err := os.Open(filename, os.O_RDONLY, 0)
    if err != nil {
        return "", err
    }
    defer f.Close()  // f.Close will run when we're finished.

    var result []byte
    buf := make([]byte, 100)
    for {
        n, err := f.Read(buf[0:])
        result = append(result, buf[0:n]...) // append is discuss
        if err != nil {
            if err == os.EOF {
                break
            }
            return "", err  // f will be closed if we return here
        }
    }
    return string(result), nil // f will be closed if we return h
}
```

这样延迟一个函数有双重优势：一是你永远不会忘记关闭文件，此错误在你事后编辑函数添加一个返回路径时常常发生。二是关闭和打开靠在一起，比放在函数尾要清晰很多。

延迟函数的参量（包括接受者，如果函数是一个方法）的求值发生在defer 语句执行时，而不是延迟函数调用时。除了不必担心函数执行时变量值的改变外，也意味着同一延迟调用可以延迟多个函数的执行。如下傻例：

```
for i := 0; i < 5; i++ {
    defer fmt.Printf("%d ", i)
}
```

延迟函数执行顺序为 LIFO，所以上面代码在函数返回时打印 4 3 2 1 0。更可信的例子是跟踪程序中函数执行的一个简单方式。我们可以写些简单的跟踪例程：

```
func trace(s string)   { fmt.Println("entering:", s) }
func untrace(s string) { fmt.Println("leaving:", s) }

// Use them like this:
func a() {
    trace("a")
    defer untrace("a")
    // do something....
}
```

利用被延迟函数的参量在 defer 执行时得值的特点，跟踪函数可以安排未跟踪函数的参量。

```
func trace(s string) string {
    fmt.Println("entering:", s)
    return s
}

func un(s string) {
    fmt.Println("leaving:", s)
}

func a() {
    defer un(trace("a"))
    fmt.Println("in a")
}

func b() {
    defer un(trace("b"))
    fmt.Println("in b")
    a()
}

func main() {
    b()
}
```

打印：

```
entering: b
in b
entering: a
in a
leaving: a
leaving: b
```

对于习惯其它语言的块层次资源管理的程序员，defer 可能比较怪，但它最有趣最强力的应用恰恰来自它不是基于块、而是基于函数。在panic 和 recover 一节我们会看到一个例子。

5.8. 数据

5.8.1. new()分配

Go 有两个分配原语，new() 和 make() 。它们做法不同，也用作不同类型上。有点乱但规则简单。我们先谈谈 new() 。它是个内部函数，本质上和其它语言的同类一样：new(T)分配一块清零的存储空间给类型 T 的新项并返回其地址，一个类型 *T 的值。 用 Go 的术语，它返回一个类型 T 的新分配的零值。

因为 new() 返回的内存清零， 可以用来安排使用零值的物件而不需再初始化。亦即数据结构的用户可以直接用 new() 生成一个并马上使用。例如， bytes.Buffer 的文档指出"零值的 Buffer 为空并可用"。同http://code.google.com/p/ac-me/ 61理，sync.Mutex 没有明确的架构函数或 init 方法。 而是，一个sync.Mutex 的零值定义为开锁的互斥。

零值有用，这个特性可以顺延。考虑下面的声明。

```
type SyncedBuffer struct {
    lock    sync.Mutex
    buffer  bytes.Buffer
}
```

类型 SyncBuffer 的值在分配或者声明后立即可用。下例，p 和 v 无需多余的安排已可以正确使用了。

```
p := new(SyncedBuffer)  // type *SyncedBuffer
var v SyncedBuffer      // type  SyncedBuffer
```

5.8.2. 构造和结构初始化

有时零值不够好，有必要使用一个初始化架构函数，如下面从 os 包引出的例子。

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }
    f := new(File)
    f.fd = fd
    f.name = name
    f.dirinfo = nil
    f.nepipe = 0
    return f
}
```

这里有很多注模。我们可用组合字面简化之，它是个每次求值即生成新实例的表达式。

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }
    f := File{fd, name, nil, 0}
    return &f
}
```

注意返回局部变量的地址是完全 OK 的；变量对应的存储空间在函数返回后仍然存在。实际上，取一个组合字面的地址使每次它求值时都生成一个新实例，因此我们可以把最后两行合起来。

```
    return &File{fd, name, nil, 0}
```

组合字面的域必须按顺序给出并全部出现。可是，明确的用域:值对儿标记元素，初始化可用任意顺序，未出现的对应着零值。所以我们可以讲

```
    return &File{fd: fd, name: name}
```

特别的，如果一个组合字面一个域也没有，它生成此类型的零值。表达式 new(File) 和 &File{} 是等价的。

组合字面也可以生成数组、切片和映射，其域为合适的下标或映射键。下例中，无论 Enone Eio 和 Einval 是什么值都可以，只要它们是不同的。

```
a := [...]string   {Enone: "no error", Eio: "Eio", Einval: "inval
s := []string      {Enone: "no error", Eio: "Eio", Einval: "inval
m := map[int]string{Enone: "no error", Eio: "Eio", Einval: "inval
```

5.8.3. make()分配

回到分配。内部函数 make(T, args) 的服务目的和 new(T) 不同。它只生成切片，映射和信道，并返回一个初始化的（不是零）的，type T的，不是 *T 的值。这种区分的原因是，这三种类型，揭开盖子，底下引用的数据结构必须在用前初始化。比如切片是一个三项的描述符，包含数据指针（数组内），长度，和容量；在这些项初始化前，切片为 nil 。对于切片、映射和信道，make 初始化内部数据结构，并准备要用的值。例如，

```
make([]int, 10, 100)
```

分配一个 100 个整数的数组，然后生成一个切片结构，长度为10，容量是100的指向此数组的首10项。（生成切片时，容量可以不写；详见切片一节。）对应的，new([]int) 返回一个新分配的，清零的切片结构，亦即，一个 nil 切片值的指针。

下面的例子展示了 new() 和 make() 的不同。

```
var p *[]int = new([]int)       // allocates slice structure; *p
var v  []int = make([]int, 100) // v now refers to a new array of

// Unnecessarily complex:
var p *[]int = new([]int)
*p = make([]int, 100, 100)

// Idiomatic:
v := make([]int, 100)
```

记住 make() 只用于映射、切片和信道，不返回指针。要明确的得到指针用 new() 分配。

5.8.4. 数组

数组用于安排详细的内存布局，还有助于避免分配，但其主要作为切片的构件，即下节的主题。这里先讲几句打个底儿。

Go 和 C 的数组的主要不同在于：

- 数组为值。数组赋值给另一数组拷贝其全部元素。
- 特别是，如果你传递数组给一个函数，它受到此数组的拷贝，不是指针。
- 数组的尺寸是其类型的一部分。[10]int 和 [20]int 是完全不同的类型。

值的属性可用但昂贵；如你所需的是类似 C 的行为和效率，你可以传递一个指针给数组。

```
func Sum(a *[3]float) (sum float) {
    for _, v := range *a {
        sum += v
    }
    return
}

array := [...]float{7.0, 8.5, 9.1}
x := Sum(&array)  // Note the explicit address-of operator
```

即便如此也不是地道的 Go 风格。切片才是。

5.8.5. Slices 切片

切片包装数组，给数据系列一个通用、强力、方便的界面。除了像变换矩阵那种要求明确尺寸的情况，绝大部分的数组编程在 Go 里使用切片、而不是简单的数组。

切片是引用类型，即如果赋值切片给另一个切片，它们都指向同一底层数组。例如，如果某函数取切片参量，对其元素的改动会显现在调用者中，类似于传递一个底层数组的指针。因此 Read 函数可以接受切片参量，而不需指针和计数；切片的长度决定了可读数据的上限。这里是 os 包的 File 型的 Read 方法的签名：

```
func (file *File) Read(buf []byte) (n int, err os.Error)
```

此方法返回读入字节数和可能的错误值。要读入一个大的缓冲 b 的首32字节， 切片（动词）缓冲。

```
    n, err := f.Read(buf[0:32])
```

这种切片常用且高效。实际上，先不管效率，此片段也可读缓冲的首32字节。

```
    var n int
    var err os.Error
    for i := 0; i < 32; i++ {
        nbytes, e := f.Read(buf[i:i+1])  // Read one byte.
        if nbytes == 0 || e != nil {
            err = e
            break
        }
        n += nbytes
    }
```

只要还在底层数组的限制内，切片的长度可以改变，只需赋值自己。切片的容量，可用内部函数 cap 取得，给出此切片可用的最大长度。下面的函数给切片添值。如果数据超过容量，切片重新分配，返回结果切片。此函数利用了 len 和 cap 对 nil 切片合法、返回0的事实。

```
func Append(slice, data[]byte) []byte {
    l := len(slice)
    if l + len(data) > cap(slice) {  // reallocate
        // Allocate double what's needed, for future growth.
        newSlice := make([]byte, (l+len(data))*2)
        // Copy data (could use bytes.Copy()).
        for i, c := range slice {
            newSlice[i] = c
        }
        slice = newSlice
    }
    slice = slice[0:l+len(data)]
    for i, c := range data {
        slice[l+i] = c
    }
    return slice
}
```

我们必须返回切片，因为尽管 Append 可以改变 slice 的元素， 切片自身（持有指针、长度和容量的运行态数据结构）是值传递的。添加切片的主意很有用，因此由内置函数 append 实现。要理解此函数的设计，我们需要多一些信息，所以稍后再讲。

5.8.6. Maps 字典

映射提供了一个方便强力的内部数据结构，用来联合不同的类型。键可以是任何定义了相等操作符的类型，如整型，浮点型，字串，指针，界面（只要其动态类型支持相等）。结构，数组和切片不可用作映射键，因为其类型未定义相等。类似切片，映射是引用类型。如果你传递映射给某函数，对映射的内容的改动显现给调用者。

映射的生成使用平常的冒号隔开的键值伴组合字面句法，所以很容易初始化时建好它们。

```
var timeZone = map[string] int {
    "UTC":  0*60*60,
    "EST": -5*60*60,
    "CST": -6*60*60,
    "MST": -7*60*60,
    "PST": -8*60*60,
}
```

赋值和获取映射值语法上就像数组，只是下标不需是整型。

```
offset := timeZone["EST"]
```

试图获取不存在的键的映射值返回对应条目类型的零值。例如，如果映射包含整型数，查找不存在的键返回0。

有时你需区分不在键和零值。 是没有 “UTC” 的条目，还是因为其值为零？你可以用多值赋值的形式加以区分。

```
var seconds int
var ok bool
seconds, ok = timeZone[tz]
```

道理很明显，此习语称为“逗号ok”。此例中，如果 tz 存在，seconds 相应赋值，ok 为真；否则，seconds 为0，ok为假。下面的函数加上了中意的出错报告：

```
func offset(tz string) int {
    if seconds, ok := timeZone[tz]; ok {
        return seconds
    }
    log.Stderr("unknown time zone", tz)
    return 0
}
```

要检查映射的存在，又不想管实际值，你可以用空白标识，即下划线（ _ ）。空白标识可以赋值或声明为任意类型的任意值，会被无害的丢弃。如只要测试映射是否存在， 在平常变量的地方使用空白标识即可。

```
_, present := timeZone[tz]
```

要删除映射条目，翻转多值赋值，在右边多放个布尔；如果布尔为假，条目被删。即便键已经不再了，这样做也是安全的。

```
timeZone["PDT"] = 0, false  // Now on Standard Time
```

5.8.7. 打印

Go 的排版打印风格类似 C 的 printf 族但更丰富更通用。这些函数活在 fmt 包里，叫大写的名字：fmt.Printf，fmt.Fprintf， fmt.Sprintf 等等。字串函数（Sprintf 等）返回字串，而不是填充给定的缓冲。

你不需给出排版字串。对应每个 Printf，Fprintf 和 Sprintf 都有另一对函数。例如 Print 和 Println。 它们不需排版字串，而是用每个参量默认的格式。Println 版本还会在参量间加入空格和输出新行，而 Print 版本只当操作数的两边都不是字串时才添加空格。下例每行的输出都是一样的：

```
fmt.Printf("Hello %d\n", 23)
fmt.Fprint(os.Stdout, "Hello ", 23, "\n")
fmt.Println(fmt.Sprint("Hello ", 23))
```

如《辅导》里所讲，fmt.Fprint 和伙伴们的第一个参量可以是任何实现 io.Writer 界面的物件。变量 os.Stdout 和 os.Stderr 是熟悉的实例。

从此事情开始偏离 C 了。首先，数字格式如 %d 没有正负和尺寸的标记；打印例程使用参量的类型决定这些属性。

```
var x uint64 = 1<<64 - 1
fmt.Printf("%d %x; %d %x\n", x, x, int64(x), int64(x))
```

打印出：

```
18446744073709551615 ffffffffffffffff; -1 -1
```

如果你只需默认的转换，例如整数用十进制，你可以用全拿格式 %v（代表 value）；结果和Print 与 Println 打印的完全一样。再有，此格式可打印任意值，包括数组，结构和映射。这里是上节定义的时区映射的打印语句。

```
fmt.Printf("%v\n", timeZone)  // or just fmt.Println(timeZone)
```

打印出：

```
map[CST:-21600 PST:-28800 EST:-18000 UTC:0 MST:-25200]
```

当然，映射的键会以任意顺序输出。打印结构时，改进的格式 %+v 用结构的域名注释，对任意值格式 %#v 打印出完整的 Go 句法。

```
type T struct {
    a int
    b float
    c string
}
t := &T{ 7, -2.35, "abc\tdef" }
fmt.Printf("%v\n", t)
fmt.Printf("%+v\n", t)
fmt.Printf("%#v\n", t)
fmt.Printf("%#v\n", timeZone)
```

打印出：

```
&{7 -2.35 abc   def}
&{a:7 b:-2.35 c:abc     def}
&main.T{a:7, b:-2.35, c:"abc\tdef"}
map[string] int{"CST":-21600, "PST":-28800, "EST":-18000, "UTC":(
```

（注意和号&）。引号括起的字串也可以 %q 用在 string 或 []byte 类型的值上，对应的格式 %#q 如果可能则使用反引号。还有，%x 可用于字串、字节数组和整型，得到长的十六进制串，有空格的格式（% x）会在字节间加空格。

另一好用的格式是 %T，打印某值的类型。

```
fmt.Printf("%T\n", timeZone)
```

打印出：

```
map[string] int
```

如果你要控制某定制类型的默认格式， 只需在其类型上定义方法String() string。对我们简单的类型 T，可以是：

```
func (t *T) String() string {
    return fmt.Sprintf("%d/%g/%q", t.a, t.b, t.c)
}
fmt.Printf("%v\n", t)
```

来打印

```
7/-2.35/"abc\tdef"
```

我们的String() 方法可以调用 Sprint，因为打印例程是完全可以重入可以递归的。我们可以更进一步，把一个打印例程的参量直接传递给另一打印例程。 Printf 的签名的首参量使用类型 ...interface{}，来指定任意数量任意类型的参量可以出现在格式字串的后面。

```
func Printf(format string, v ...) (n int, errno os.Error) {
```

Printf 函数中，v 像是一个 []interface{} 类的变量。但如果把它传递给另一个多维函数，它就像一列普通的参量。这里是我们上面用过的log.Println 的实现。它把自己的参量直接传递给 fmt.Sprintln 来实际打印。

```
// Stderr is a helper function for easy logging to stderr. It is
func Stderr(v ...) {
    stderr.Output(2, fmt.Sprintln(v))  // Output takes parameters
}
```

我们在 Sprintln 的调用的 v 后写 ... 告诉编译器把 v 作为一列参量；否则它只是传递一个单一的切片参量。

还有很多打印的内容我们还没讲，细节可参考 godoc 的 fmt 包的文档。

顺便提一句， ... 参量可以是任意给定的类型，例如，...int 在 min 函数里可以选一列整数的最小值。

```
func Min(a ...int) int {
    min := int(^uint(0) >> 1)  // largest int
    for _, i := range a {
        if i < min {
            min = i
        }
    }
    return min
}
```

5.8.8. Append

现在我们解释 append 的设计。append 的签名和上面我们定制的Append 函数不同。大体上是：

```
func append(slice []T, elements...T) []T
```

T 替代的是任意类型。 实际中你不能写 Go 的函数由调用者决定 T 的类型，所以 append 内置：它需要编译器的支持。

append 所做的是在切片尾添加元素并返回结果。结果需要返回因为，正如我们手写的 Append，底层的数组可能更改。下面简单的例子：

```
x := []int{1,2,3}
x = append(x, 4, 5, 6)
fmt.Println(x)
```

打印 1 2 3 4 5。所以 append 有点像 Printf 收集任意数量的参量。

但如何像我们 Append 一样给切片添加切片呢？容易：使用 ... 在调用的地方，正如我们上面我们调用 Output。 下例产生如上同样的输出：

```
x := []int{1,2,3}
y := []int{4,5,6}
x = append(x, y...)
fmt.Println(x)
```

没有 ... 将不能编译，因为类型错误； y 不是 int 类型。

5.9. 初始化

尽管表面看来和 C 或 C++ 的初始化没什么不同，Go 的更够强。复杂结构可在初始化时架设，并且不同包的物件的初始化顺序问题也得到正确处理。

5.9.1. Constants 常量初始化

常量在 Go 里是 —— 不变的。它们在编译时生成，即便是局部定义在函数里。它只能是数，字串或布尔。因为编译态的限制，定义它们的表达式必须是常量表达式，可以被编译器求值。例如，1<<3 是常量表达式， math.Sin(math.Pi/4) 不是，因为 math.Sin 的函数调用发生在运行态。

Go 的列举常量可用 iota 生成。 因为 iota 可以是表达式的一部分，并且表达式可以隐含重复，打造一套精致的值可以变得很容易。

```
type ByteSize float64
const (
    _ = iota  // ignore first value by assigning to blank identif
    KB ByteSize = 1<<(10*iota)
    MB
    GB
    TB
    PB
    EB
    ZB
    YB
)
```

给类型添加比如 String 等方法的本领，使值自动排版打印自己变得可能，即使只作为通用类型的一部分。

```
func (b ByteSize) String() string {
    switch {
    case b >= YB:
        return fmt.Sprintf("%.2fYB", b/YB)
    case b >= ZB:
        return fmt.Sprintf("%.2fZB", b/ZB)
    case b >= EB:
        return fmt.Sprintf("%.2fEB", b/EB)
    case b >= PB:
        return fmt.Sprintf("%.2fPB", b/PB)
    case b >= TB:
        return fmt.Sprintf("%.2fTB", b/TB)
    case b >= GB:
        return fmt.Sprintf("%.2fGB", b/GB)
    case b >= MB:
        return fmt.Sprintf("%.2fMB", b/MB)
    case b >= KB:
        return fmt.Sprintf("%.2fKB", b/KB)
    }
    return fmt.Sprintf("%.2fB", b)
}
```

表达式 YB 打印 1.00YB，而 ByteSize(1e13) 打印 9.09TB

5.9.2. 变量初始化

变量和常量的初始化一样，但可以用运行态计算的普通表达式。

```
var (
    HOME = os.Getenv("HOME")
    USER = os.Getenv("USER")
    GOROOT = os.Getenv("GOROOT")
)
```

5.9.3. init函数

最后，每个源文件可以定义自身的 init() 函数来安排所需的状态。唯一的限制是，尽管够程可在初始化时启动，它们只在初始完成后执行；初始化永远是单一的执行序列。最后之后，init() 发生在包里所有变量初始化之后，而其又发生在所有的包全部导入之后。

除了初始化不能表示为声明外，init() 函数常用来在程序运行前验证或修补其状态。

```
func init() {
    if USER == "" {
        log.Exit("$USER not set")
    }
    if HOME == "" {
        HOME = "/usr/" + USER
    }
    if GOROOT == "" {
        GOROOT = HOME + "/go"
    }
    // GOROOT may be overridden by --goroot flag on command line
    flag.StringVar(&GOROOT, "goroot", GOROOT, "Go root directory"
}
```

5.10. 方法

5.10.1. 指针vs值

方法可用于任意带名的非指针和界面的类型；接受者没必要是结构。

在上面讨论切片时，我们写了个 Append 函数。其实我们可以把它定义为切片的方法。首先我们声明一个带名的类型，以便我们在其上施加方法，并使此方法的接受者的值是此类型。

```
type ByteSlice []byte

func (slice ByteSlice) Append(data []byte) []byte {
    // Body exactly the same as above
}
```

这仍需方法返回更新的切片，更灵活的方式是定义方法接受 ByteSlice 的指针，以便重写调用着的切片。

```
func (p *ByteSlice) Append(data []byte) {
    slice := *p
    // Body as above, without the return.
    *p = slice
}
```

实际上，我们能做的更好。如果我们把函数改的像标准的 Write 方法，例如：

```
func (p *ByteSlice) Write(data []byte) (n int, err os.Error) {
    slice := *p
    // Again as above.
    *p = slice
    return len(data), nil
}
```

此时类型 *ByteSlice 可以满足标准界面 io.Writer，很方便的，例如我们打印到里面：

```
    var b ByteSlice
    fmt.Fprintf(&b, "This hour has %d days\n", 7)
```

我们传递 ByteSlice 的地址是因为只有 *ByteSlice 满足 io.Writer。接受者的指针和值规则是，值的方法可用于指针和值，而指针的方法只用于指针。这是因为指针方法可以改变接受者；使用拷贝的值会导致这些改变的丢失。

顺便一提，字节切片的 Write 已在 bytes.Buffer 实现。

5.11. 接口和其他类型

5.11.1. 接口

Go中的接口提供了一类对象的抽象。我们在前面已经看到了关于接口的一些例子。我们可以给新定义的对象实现一个String方法，这样就可以用 Fprintf输出该类型的值。同样，Fprintf可以将 结果输出到任意实现了Write方法的对象。接口一般只包含一类方法， 并且以ed后缀的方式命名，例如io.Writer接口对应Write 方法实现。

一种类型可以实现多个接口。例如，如果想支持sort包中的排序 方法，那么只需要实现sort.Interface接口的Len()、 Less()、Swap(i, j int)方法就可以了。下面的例子实现了sort.Interface接口的同时，还定制了输出函数。

```
type Sequence []int

// Methods required by sort.Interface.
func (s Sequence) Len() int {
    return len(s)
}
func (s Sequence) Less(i, j int) bool {
    return s[i] < s[j]
}
func (s Sequence) Swap(i, j int) {
    s[i], s[j] = s[j], s[i]
}

// Method for printing - sorts the elements before printing.
func (s Sequence) String() string {
    sort.Sort(s)
    str := "["
    for i, elem := range s {
        if i > 0 {
            str += " "
        }
        str += fmt.Sprint(elem)
    }
    return str + "]"
}
```

5.11.2. 转换

Sequence 的 String 方法重做了 Sprint 在切片的工作。我们可以在调用 Sprint 前把 Sequence 转为普通的 []int。

```
func (s Sequence) String() string {
    sort.Sort(s)
    return fmt.Sprint([]int(s))
}
```

转换使得 s 被当作普通的切片，因此得到默认的排版。不加转换，Sprint 会发现 Sequence 的 String 方法，进而无穷递归。如果忽略类型名称，Sequence 和 []int 的类型相同， 它们之间的转换是合法的。转换不会得到新值，它只是暂时假装现有值是新类型。（其它合法的转换，如从整型到浮点型，会生成新值。）

地道的 Go 程序会转换表达式的类型来使用一组不同的方法。例如，我们可以用现有类型 sort.IntArray 把整个例子缩减为：

```
type Sequence []int

// Method for printing - sorts the elements before printing
func (s Sequence) String() string {
    sort.IntArray(s).Sort()
    return fmt.Sprint([]int(s))
}
```

现在，无需让 Sequence 实现多个界面（排序和打印），我们使用了把数据转换为多种类型（Sequence，sort.IntArray 和 []int）的能力，每个来完成一部分的工作。这实际上不常见但很有效。

5.11.3. Generality(泛化)

如果某类型的存在只为了实现某界面，而除此之外没有其它导出的方法，则此类型也不需导出。只导出界面明确了只有行为有价值，而不是实现，其它的不同特性的实现可以镜像其原来的类型。这样也避免了为同一方法的不同实现做文档。

此时，架构函数应返回界面而不是实现类型。例如，哈希库的crc32.NewIEEE() 和 adler32.New() 都返回界面类型 hash.Hash32。 替换一个 Go 出现的 CRC-32 算法为 Adler-32 只需改变架构函数的调用；其余的代码不受算法改变的影响。

同样的方式使得 crypto/block 包的流密（streaming cipher）算法与链接在一起的块密（block cipher）相区隔。比对 bu?o 包，它们包装了界面，返回 hash.Hash，io.Reader 和 io.Writer 界面值，而不是特定的实现。

crypto/block 界面包括：

```
type Cipher interface {
    BlockSize() int
    Encrypt(src, dst []byte)
    Decrypt(src, dst []byte)
}

// NewECBDecrypter returns a reader that reads data
// from r and decrypts it using c in electronic codebook (ECB) mo
func NewECBDecrypter(c Cipher, r io.Reader) io.Reader

// NewCBCDecrypter returns a reader that reads data
// from r and decrypts it using c in cipher block chaining (CBC)
// with the initialization vector iv.
func NewCBCDecrypter(c Cipher, iv []byte, r io.Reader) io.Reader
```

NewECBDecrypter 和 NewCBCReader 不只用于某特定的加密算法和数据源，而是任意的 Cipher 界面的实现和任意的 io.Reader。因为它们返回 io.Reader 界面值，替换 ECB 加密为 CBC 加密只是局部修改。 架构函数必须编辑，但因为周围代码必须只把结果作为io.Reader，它不会注意到有什么不同。

5.11.4. 接口和方法

因为几乎任何东西都可加以方法，几乎任何东西都可满足某界面。一个展示的例子是 http 包定义的 Handler 界面。任何物件实现了Handler 都可服务 HTTP 请求。

```
type Handler interface {
    ServeHTTP(*Conn, *Request)
}
```

ResponseWriter 本身是个界面，它提供一些可访问的方法来返回客户的请求。这些方法包括标准的 Write 方法。因此 http.ResponseWriter 可用在 io.Writer 可以使用的地方。Request 是个结构，包含客户请求的一个解析过的表示。

为求简短，我们忽略 POST 并假定所有 HTTP 请求都是 GET；此简化不会影响经手者的设置。下面一个小而全的经手者实现了网页访问次数的计数。

```
// Simple counter server.
type Counter struct {
    n int
}

func (ctr *Counter) ServeHTTP(c *http.Conn, req *http.Request) {
    ctr.n++
    fmt.Fprintf(c, "counter = %d\n", ctr.n)
}
```

（注意 Fprintf 怎样打印到 http.ResponseWriter）。作为参考，这里是怎样把服务者加在一个 URL 树的节点上。

```
import "http"
...
ctr := new(Counter)
http.Handle("/counter", ctr)
```

可是为何把 Counter 作为结构呢？一个整数能够了。（接受者需是指针，使增量带回调用者）。

```
// Simpler counter server.
type Counter int

func (ctr *Counter) ServeHTTP(c *http.Conn, req *http.Request) {
    *ctr++
    fmt.Fprintf(c, "counter = %d\n", *ctr)
}
```

当某页被访问时怎样通知你的程序更新某些内部状态呢？给网页贴个信道。

```
// A channel that sends a notification on each visit.
// (Probably want the channel to be buffered.)
type Chan chan *http.Request

func (ch Chan) ServeHTTP(c *http.Conn, req *http.Request) {
    ch <- req
    fmt.Fprint(c, "notification sent")
}
```

最后，让我们在 /args 显示启动服务器时的参量。写个打印参量的函数很容易：

```
func ArgServer() {
    for i, s := range os.Args {
        fmt.Println(s)
    }
}
```

怎样把它变成 HTTP 服务器呢？我们可以把 ArgServer 作为某个类型的方法再忽略其值，也有更干净的做法。既然我们可以给任意非指针和界面的类型定义方法，我们可以给函数写个方法。http 包里有如下代码：

```
// The HandlerFunc type is an adapter to allow the use of
// ordinary functions as HTTP handlers.  If f is a function
// with the appropriate signature, HandlerFunc(f) is a
// Handler object that calls f.
type HandlerFunc func(*Conn, *Request)

// ServeHTTP calls f(c, req).
func (f HandlerFunc) ServeHTTP(c *Conn, req *Request) {
    f(c, req)
}
```

HandlerFunc 是个带 ServeHTTP 方法的类型， 所以此类的值都可以服务 HTTP 请求。我们来看看此方法的实现：接受者是个函数，f，方法调用 f 。看起来很怪，但和，比如，接受者是信道，而方法发送到 此信道，没什么不同。

要把 ArgServer 变为 HTTP 服务器， 我们首先改成正确的签名：

```
// Argument server.
func ArgServer(c *http.Conn, req *http.Request) {
    for i, s := range os.Args {
        fmt.Fprintln(c, s)
    }
}
```

ArgServer 现在和 HandlerFunc 有同样的签名，就可以转成此类使用其方法，就像我们把 Sequence 转为 IntArray 来使用 IntArray.Sort 一样。设置代码很简短：

```
http.Handle("/args", http.HandlerFunc(ArgServer))
```

当有人访问 /args 页时，此页的经手者有值 ArgServer 和类型HandlerFunc。HTTP 服务器启动此类型的 ServeHTTP 方法，用ArgServer 作为接受者，反过来调用 ArgServer （通过启动handlerFunc.ServeHTTP 的 f(w, req) 。）参量被显示出来。

此节中我们从一个结构，整数，信道和一个函数制造出一个 HTTP 服务器，全赖于界面就是一套方法，可定义在（几乎）任何类型上。

5.12. 内置

Go 不提供通行的、类型驱动的子类划分的概念，但它通过在结构或界面内置，确实有能力从某实现“借”些片段。

界面内置非常简单。我们提到过 io.Reader 和 io.Writer 的界面；这里是其定义：

```
type Reader interface {
    Read(p []byte) (n int, err os.Error)
}

type Writer interface {
    Write(p []byte) (n int, err os.Error)
}
```

io 包也导出一些其它的界面来规范实现其多个方法的物件。例如，io.ReadWriter 界面同时包括 Read 和 Write 。我们可以明确的列出这两个方法来规定 io.ReadWriter ，但更简单更有启发的是内置这两个界面形成新界面，如下：

```
// ReadWrite is the interface that groups the basic Read and Writ
type ReadWriter interface {
    Reader
    Writer
}
```

顾名思意，ReadWriter 可做 Reader 和 Writer 两个的事；它是内置界面的集合（必须是没有交集的方法）。只有界面可以内置在界面里。

同样的基本概念适用于结构，但牵连更广。bu?o 包有两个结构类型，bu?o.Reader 和 bu?o.Writer ，每个都当然对应实现着 io 包的界面。并且，bu?o 也实现了缓冲的读写，这是通过把 reader 和 writer 内置到一个结构做到的；它列出类型但不给名称：

```
// ReadWriter stores pointers to a Reader and a Writer.
// It implements io.ReadWriter.
type ReadWriter struct {
    *Reader  // *bufio.Reader
    *Writer  // *bufio.Writer
}
```

内置元素是指针所以使用前必须初始化指向有效的结构。 ReadWriter 结构可以写为：

```
type ReadWriter struct {
    reader *Reader
    writer *Writer
}
```

但要提升域的方法又要满足 io 界面，我们还需提供转发方法，如：

```
func (rw *ReadWriter) Read(p []byte) (n int, err os.Error) {
    return rw.reader.Read(p)
}
```

通过直接内置结构，我们避免了这些账目管理。内置类型的方法是附送的，亦即 bu?o.ReadWriter 除了有 bu?o.Reader 和bu?o.Writer 的方法，还同时满足三个界面：io.Reader，io.Writer 和 io.ReadWriter 。

内置和子类划分有着重要不同。我们内置类型时，此类的方法成为外层类型的方法，但调用时其接受者是内层类型，而不是外层。我们的例子里，当 bu?o.ReadWriter 的 Read 方法被调用时，它的效果和上面所写的转发方法完全一样；接受者是 ReadWriter 的 reader，而不是 ReadWriter 自身。

内置也用来提供某种便利。下例是一个内置域和普通的，带名的域在一起：

```
type Job struct {
    Command string
    *log.Logger
}
```

此时 Job 类型有 Log，Logf 和其它 *log.Logger 的方法。 我们当然可以给 Logger 个名字，但无此必要。这里，初始化后，我们可以 log Job：

```
job.Log("starting now...")
```

Logger 是结构的普通域所以我们能用通常的架构函数初始化它：

```
func NewJob(command string, logger *log.Logger) *Job {
    return
&Job{command, logger}
}
```

或使用组合字面：

```
job := &Job{command, log.New(os.Stderr, nil, "Job: ", log.Ldate)
```

如果我们需要直接引用内置域，域的类型名，忽略包标识，可作为域名。如果我们要得到 Job 变量 job 的 *log.Logger，我们用job.Logger 。这可用在细化 Logger 的方法上：

```
func (job *Job) Logf(format string, args ...) {
    job.Logger.Logf("%q: %s", job.Command, fmt.Sprintf(format, ar
}
```

内置类型导致撞名的问题，但解决方案很简单。首先，域或方法 X 隐藏此类型更深层部分的 X 项。如果 log.Logger 包括叫 Command 的域或方法，则 Job 的 Command 域占主导权。

其次，如果同层出现同名，通常是个错误；如果 Job 结构有另一域或方法叫 Logger，要内置 log.Logger 会出错。但只要重名在类型定义外的程序中不被提及，就不成问题。此条件提供了改动外部的内置类型的某种保护；只要两个域都没被用到，新增的域名和另一子类的域重名也没有问题。

5.13. 并发

5.13.1. 交流来分享

并发编程是很大的主题，此处只够讲 Go 方面的要点。

很多环境的并发编程变得困难出自于实现正确读写共享变量的微妙性。Go 鼓励一种不一样的方式，这里，共享变量在信道是传递，并且事实上，从来未被独立的执行序列所共享。每一特定时间只有一个够程在存取该值。从设计上数据竞争就不会发生。为鼓励这种思考方式我们把它缩减成一句口号：

> 别靠共享内存来通信，要靠通信来分享内存!

此方式可扯的太远。例如，引用计数最好靠整型变量外加一个互斥。但最为高层方式，使用信道控制存取可以更容易写成清楚正确的程序。

思考这种模型一种方式是考虑在单一 CPU 上运行的典型的单线程程序，不需用到同步原语。现在多运行一份；也同样不需同步。现在让两者通信；如果通信是同步者，还是不需其它的同步。例如，Unix 的管道完美的适合这个模型。尽管 Go 的并行方式源自 Hoare 的通信顺序进程（CSP），它也可视为泛化的类型安全的 Unix 管道。

5.13.2. Goroutines(Go程)

它们叫做够程， 是因为现有的术语 -- 线程，协程和进程等 -- 传达的含义不够精确。够程的模式很简单：它是在同一地址空间和其它够程并列执行的函数。它轻盈，只比分配堆栈空间多费一点儿。因为堆栈开始时很小，所以它们很便宜，只在需要时分配（和释放）堆库存。

够程在多个 OS 线程间复用，所以如果某个需要阻塞，例如在等待IO，其它的可继续执行。它们的设计隐藏了许多线程生成和管理的复杂性。

在某个函数或方法前加上 go 键字则在新够程中执行此调用。当调用完成，够程安静的退出。（效果类似 Unix shell 的 & -- 在后台执行命令。）

```
go list.Sort()  // run list.Sort in parallel; don't wait for it.
```

函数字面在实施够程上很顺手：

```
func Announce(message string, delay int64) {
    go func() {
        time.Sleep(delay)
        fmt.Println(message)
    }()  // Note the parentheses - must call the function.
}
```

在 Go 里，函数字面是闭包：实现保证函数引用的变量可以活到它们不在用了。

这些例子没什么用处，因为函数无法通知其结束。那需用到信道。

5.13.3. Channels(信道)

类型映射，信道是引用类型，使用 make 分配。如果提供了可选的整型参量，它会设置信道的缓冲大小。默认是0，即无缓冲的或同步的信道。

```
ci := make(chan int)            // unbuffered channel of integers
cj := make(chan int, 0)         // unbuffered channel of integers
cs := make(chan *os.File, 100)  // buffered channel of pointers t
```

信道结合了通信 -- 即值的交换 -- 和同步 -- 确保两个计算（够程）处于某个已知状态。

信道有很多惯用语。先从一个开始。上节我们启动了个后台的排序。信道使启动够程能等待排序完成。

```
c := make(chan int)  // Allocate a channel.
// Start the sort in a goroutine; when it completes, signal on th
go func() {
    list.Sort()
    c <- 1  // Send a signal; value does not matter.
}()
doSomethingForAWhile()
<-c   // Wait for sort to finish; discard sent value.
```

接收者阻塞到有数据可以接收。如果信道是非缓冲的，发送者阻塞到接收者收到其值。如果信道有缓冲，发送者只需阻塞到值拷贝到缓冲里；如果缓冲满，则等待直到某个接收者取走一值。

一个缓冲信道可以用作信号灯，比如用来限速。下例中，到来请求传递给 handle，来发送一值到信道，处理请求，以及才信道接收值。信道的容量决定了可同时调用 process 的数量。

```
var sem = make(chan int, MaxOutstanding)

func handle(r *Request) {
    sem <- 1    // Wait for active queue to drain.
    process(r)  // May take a long time.
    <-sem       // Done; enable next request to run.
}

func Serve(queue chan *Request) {
    for {
        req := <-queue
        go handle(req)  // Don't wait for handle to finish.
    }
}
```

同样的概念，我们可以启动一定数量的 handle 够程，全都读取请求信道。够程的数量限制同时调用 process 的数量。此 Serve 函数也接受一个信道来告知它退出；够程启动后会接收阻塞在此信道。

```
func handle(queue chan *Request) {
    for r := range queue {
        process(r)
    }
}

func Serve(clientRequests chan *clientRequests, quit chan bool) {
    // Start handlers
    for i := 0; i < MaxOutstanding; i++ {
        go handle(clientRequests)
    }
    <-quit  // Wait to be told to exit.
}
```

5.13.4. Channels of channels(信道的信道)

Go 的一个最重要的特色是信道作为一等值可以被分配被传递，正如其它的值。此特色常用来实现安全并行的分路器。

上节的例子里，handle 是个理想化的请求经手者，但我们并未定义其经手的类型。如果此类型包括一个可回发的信道，每个客户都可提供自身的回答途径。下面是类型 Request 的语义定义。

```
type Request struct {
    args        []int
    f           func([]int) int
    resultChan  chan int
}
```

客户提供一个函数及其参量，以及在请求物件里的一个信道，用来接收答案。

```
func sum(a []int) (s int) {
    for _, v := range a {
        s += v
    }
    return
}

request := &Request{[]int{3, 4, 5}, sum, make(chan int)}
// Send request
clientRequests <- request
// Wait for response.
fmt.Printf("answer: %d\n", <-request.resultChan)
```

服务器端，经手函数是唯一需要改变的。

```
func handle(queue chan *Request) {
    for req := range queue {
        req.resultChan <- req.f(req.args)
    }
}
```

当然还要很多工作使其实际，但此代码是个速率限制、并行、非阻塞的 RPC 系统的框架，而且看不到一个互斥。

5.13.5. 并发

这些概念的另一应用是在多 CPU 核上并发计算。如果一个运算可以分解为独立片段，则可并发，用一信道通知每个片段的结束。

比如我们有个很花时间的运算执行在一列项上，每个项的运算值都是独立的，如下例：

```
type Vector []float64

// Apply the operation to v[i], v[i+1] ... up to v[n-1].
func (v Vector) DoSome(i, n int, u Vector, c chan int) {
    for ; i < n; i++ {
        v[i] += u.Op(v[i])
    }
    c <- 1    // signal that this piece is done
}
```

我们在循环里单独启动片段，每个 CPU 一个。 它们谁先完成都没关系；我们只是启动全部够程前清空信道，再数数结束通知即可。

```
const NCPU = 4  // number of CPU cores

func (v Vector) DoAll(u Vector) {
    c := make(chan int, NCPU)  // Buffering optional but sensible
    for i := 0; i < NCPU; i++ {
        go v.DoSome(i*len(v)/NCPU, (i+1)*len(v)/NCPU, u, c)
    }
    // Drain the channel.
    for i := 0; i < NCPU; i++ {
        <-c    // wait for one task to complete
    }
    // All done.
}
```

现在的 gc 实现（6g 等）不会默认的并发此代码。它只投入一个核给用户层的运算。任意多的够程可以阻塞在系统调用上，但默认的每个时刻只有一个可以执行用户层的代码。它本该更聪明些，某天它会变聪明，但那之前如果你要并发 CPU 就必须告知运行态你要同时执行代码的够程的数量。有两种相关的办法，或者你把环境变量GOMAXPROCS 设为你要用到的核数（默认1）；或者导入 runtime 包调用 runtime.GOMAXPROCS(NCPU)。再提一遍，此要求会随着调度及运行态的进步而退休。

5.13.6. 漏水缓冲

并发编程的工具也可用来使非并发的概念更容易表达。下例是从某个RPC 包里提取的。客户够程循环接收数据自某源，可能是网络。为免分配释放缓冲，它保有一个自由列，并由一个缓冲的信道代表。如果信道空，则新缓冲被分配。当消息缓冲好时，它在 serverChan 上发给服务器。

```
var freeList = make(chan *Buffer, 100)
var serverChan = make(chan *Buffer)

func client() {
    for {
        b, ok := <-freeList  // grab a buffer if available
        if !ok {              // if not, allocate a new one
            b = new(Buffer)
        }
        load(b)              // read next message from the net
        serverChan <- b      // send to server
    }
}
```

服务器循环读消息自客户，处理，返回缓冲到自由列。

```
func server() {
    for {
        b := <-serverChan    // wait for work
        process(b)
        _ = freeList <- b    // reuse buffer if room
    }
}
```

客户无阻的从 freeList 得到一个缓冲，如还没有则客户分配个新的缓冲。服务器无阻的发送给 freeList，放 b 回自由列，除非列满，此时缓冲掉到地板上被垃圾收集器回收。（发送操作赋值给空白标识使其 无阻但会忽略操作是否成功。）此实现仅用几行就打造了个漏水缓冲，靠缓冲信道和垃圾收集器记账。

5.14. 错误处理

一些函数在调用后一般会返回一些错误的标志。在Go中我们可以用返回返回多个值来 方便地处理错误标志信息。一般情况下，错误都实现了os.Error 接口。

```
type Error interface {
    String() string
}
```

库的编写者一般会在os.Error接口的基础上扩展更多的信息，这样函数调用者 可以知道错误的更多细节。例如：os.Open返回的是os.PathError 类型错误（里面已经包含最基本的错误接口）。

```
// PathError records an error and the operation and
// file path that caused it.
type PathError struct {
    Op string    // "open", "unlink", etc.
    Path string  // The associated file.
    Error Error  // Returned by the system call.
}

func (e *PathError) String() string {
    return e.Op + " " + e.Path + ": " + e.Error.String()
}
```

PathError生成的String错误信息如下：

```
open /etc/passwx: no such file or directory
```

这个错误信息包含了要操作的文件名，对文件的具体操作，以及操作系统返回的错误信息。 这样肯定比简单输出"no such file or directory"错误信息更有价值。

如果函数调用者想获取错误的全部细节，那么需要将错误结果从基本的类型动态转换到 更具体的错误类型。例如：下面的代码将Error转换为PathErrors 类型，因为后者的错误细细更加丰富。

```
for try := 0; try < 2; try++ {
    file, err = os.Open(filename, os.O_RDONLY, 0)
    if err == nil {
        return
    }
    if e, ok := err.(*os.PathError); ok && e.Error == os.ENOSPC {
        deleteTempFiles()  // Recover some space.
        continue
    }
    return
}
```

5.14.1. Panic(怕死)

通常报错的方式是给调用者一个多于的 os.Error 的返回值。经典的Read 方法是个出名的实例；它返回字节数和 os.Error 。但错误不可恢复则如何？有时程序就是不可再继续了。

基于此目的，内部函数 panic 实际上会生成一个运行态错误来终止程序（但参见下节）。此函数取一个任意类型的参量 —— 通常是字串——在程序死掉时打印。它也用来指出某种不可能的事情发生了，例http://code.google.com/p/ac-me/ 99如从永久循环中退出了。实际上，编辑器看到函数尾的 panic 会压制通常的 return 语句检查。

```
// A toy implementation of cube root using Newton's method.
func CubeRoot(x float64) float64 {
    z := x/3   // Arbitrary intitial value
    for i := 0; i < 1e6; i++ {
        prevz := z
        z -= (z*z*z-x) / (3*z*z)
        if veryClose(z, prevz) {
            return z
        }
    }
    // A million iterations has not converged; something is wrong
    panic(fmt.Sprintf("CubeRoot(%g) did not converge", x))
}
```

这只是个例子，实际的库函数要避免使用 panic 。如果某问题可被屏蔽或绕过，最好让事情继续而不是打死整个程序。一个可能的反例是在初始化时，如果某库函数怎么都不能安排好自己，有理由 panic，可以这么说。

```
var user = os.Getenv("USER")

func init() {
    if user == "" {
        panic("no value for $USER")
    }
}
```

5.14.2. Recover(回生)

当 panic 被叫，包括运行态错误例如数组下标越界或类型断言失败时，它会立即停止当前函数的执行，并开始退绕够程的堆栈，随之运行所有的延迟函数。如果退绕到够程堆栈顶，程序死掉。但是，我们可以用内部函数 recover 重新控制够程，恢复正常运行。

recover 的调用终止退绕并返回传给 panic 的参量。因为退绕时只有延迟函数的代码在运行，recover 只在延迟函数有用。

recover 的一个用途是在服务器内关闭失败的够程而不会杀死其它正在运行的够程。

```
func server(workChan <-chan *Work) {
    for work := range workChan {
        go safelyDo(work)
    }
}

func safelyDo(work *Work) {
    defer func() {
        if err := recover(); err != nil {
            log.Stderr("work failed:", err)
        }
    }()
    do(work)
}
```

此例子中，如果 do(work) panic了，结果会记录下，够程会不扰人的干净地退出。没必要在延迟函数做其它的事；recover 的调用完全可以处理。

意有了这种复原的模式，do 函数（及其所有的调用）可以用 panic 从任何糟糕的情况里脱身。我们可用此概念简化复杂软件的出错处理。我们看看 regexp 包里一个理想化的节选，它用局部的 Error 类型调用 panic 来报错。 下面是 Error，error 方法，和 Compile 函数的定义：

```
// Error is the type of a parse error; it satisfies os.Error.
type Error string
func (e Error) String() string {
    return string(e)
}

// error is a method of *Regexp that reports parsing errors by
// panicking with an Error.
func (regexp *Regexp) error(err string) {
    panic(Error(err))
}

// Compile returns a parsed representation of the regular express
func Compile(str string) (regexp *Regexp, err os.Error) {
    regexp = new(Regexp)
    // doParse will panic if there is a parse error.
    defer func() {
        if e := recover(); e != nil {
            regexp = nil    // Clear return value.
            err = e.(Error) // Will re-panic if not a parse error
        }
    }()
    return regexp.doParse(str), nil
}
```

如果 doParse panic了，复原块会设置返回值为 nil ——延迟函数可以修改带名的返回值。它然后通过断定 err 的赋值是类型 Error 来检查问题出自语法分析。如果不是，类型断言会失败，导致一个运行态错误，继续堆栈退绕，就好像无事发生一样。这个检查意味着如果未曾预料的事情发生了，例如数组下标越界，代码会失败，尽管我们用了panic 和 recover 出来用户触发的错误。

有了这种出错处理，error 方法能轻易的报错，而不需担心自己动手退绕堆栈。

这种有用的模式只应在一个包的内部使用。 Parse 将其内部的 panic 调用转为 os.Error 值；不把 panic 暴露给客户。这个好规则值得效法。

5.15. Web服务器

现在让我们来实现一个完整的程序：一个简单的web服务器。这其实是一个转发服务器。 google的http://chart.apis.google.com 提供了一个将数据转换为图表的服务。不过那个图表的转换程序使用比较复杂，因为需要用户 自己设置各种参数。不过我们这里的程序界面要稍微友好一点：因为我们只需要获取一小段数据， 然后调用google的图表转换程序生存QR码（Quick Response缩写，二维条码）， 对于文本 信息下编码。二维条码图像可以用手机上的摄像机采集，然后解析得到解码后的信息。

下面是完整的程序：

```
package main

import (
    "flag"
    "http"
    "io"
    "log"
    "strings"
    "template"
)

var addr = flag.String("addr", ":1718", "http service address") /
var fmap = template.FormatterMap{
    "html": template.HTMLFormatter,
    "url+html": UrlHtmlFormatter,
}
var templ = template.MustParse(templateStr, fmap)

func main() {
    flag.Parse()
    http.Handle("/", http.HandlerFunc(QR))
    err := http.ListenAndServe(*addr, nil)
    if err != nil {
        log.Exit("ListenAndServe:", err)
    }
}

func QR(c *http.Conn, req *http.Request) {
    templ.Execute(req.FormValue("s"), c)
}

func UrlHtmlFormatter(w io.Writer, v interface{}, fmt string) {
    template.HTMLEscape(w, strings.Bytes(http.URLEscape(v.(string
}

const templateStr = `
<html>
```

```
        <head>
            <title>QR Link Generator</title>
        </head>
    <body>
        {.section @}
        <img src="http://chart.apis.google.com/chart?chs=300x300&cht=
    />
        <br>
        {@|html}
        <br>
        <br>
        {.end}
        <form action="/" name=f method="GET">
            <input maxLength=1024 size=70 name=s value="" title="Text
            <input type=submit value=&##34;Show QR" name=qr>
        </form>
    </body>
    </html>
    `
```

main函数的开始部分比较简单。有一个flag选项用于指定HTTP服务器的监听端口。还有一个模板变量templ，主要用于保存HTML页面的生成模板，我们稍后会讨论。

main首先分析命令行选项，然后帮定QR函数为服务器的根目录 处理函数。最后http.ListenAndServe启动服务器，并在服务器运行期间一直 阻塞。

QR接收客户端请求，然后用表单中的s变量的值替换到模板。

template包实现了json-template。 我们的程序的简洁正是得益于template包的强大功能。本质上，在执行templ.Execute的 时候，根据需要替换调模板中的某些区域。这里的原始模板文本保存在templateStr中， 其中花括弧部分对应模板的动作。在{.section @}和{.end}之间的 以@开头的元素，在处理模板的时候会被替换。

标记{@|url+html}的意思是在格式化模板的时候，用格式化字典（fmap）中"url+html" 关键字对应的函数的处理标签的替代文本。这里的UrlHtmlFormatter函数，只是为了安全启见过滤包含的不合法信息。

这里的模板只是用于显式的html页面。如果觉得上面的解释比较简略的话，可以看到template包的 documentation。

我们仅仅用很少的代码加一些HTML文本就实现了一个有意思的webserver。使用go，往往用很少的 代码就能实现强大的功能。

6. 如何编写Go程序

6.1. 简介

本文档会介绍如何编写一个新的包，以及如何测试代码。本文档假设读者已经根据安装指南成功地安装了Go。

在着手修改已有的包或是新建包之前，一定要先把自己的想法发到邮件列表。

6.2. 社区资源

寻求实时帮助，可以在Freenode IRC服务器的#go-nuts频道里找到其他的用户或是开发人员。

Go语言的官方邮件列表是Go Nuts。

报告Bug可以使用Go问题追踪器。

对于想及时了解开发进度的读者，可以加入另一个邮件列表golang-chenkins，这样在有人往Go代码库中checkin新代码时就会收到一封简要的邮件。

6.3. 新建一个包

根据一般约定，导入路径为x/y的包的源代码应放在$GOROOT/src/pkg/x/y目录中。

6.3.1. Makefile

如果能有专门针对Go的工具能检测源代码文件，决定编译顺序就好了，但现在，我们还只能用GNU的make。所以，新建包首先要新建的文件就是Makefile。如果是在Go源代码树中，其基本格式可参照src/pkg/container/vector/Makefile：

```
include ../../../Make.inc
TARG=container/vector
GOFILES=      intvector.go      stringvector.go      vector.go  i
```

在Go的源代码树之外（个人包），标准的格式则是：

```
include $(GOROOT)/src/Make.inc
TARG=mypackage
GOFILES=      my1.go      my2.go  include $(GOROOT)/src/Make.pkg
```

第一行和最后一行分别导入了标准定义和规则。Go源代码树中所维护的包使用相对路径（代替$(GOROOT)/src），所以即使是$(GOROOT) 中含有空格也可以正常使用。这无疑简化了程序员尝试Go的难度。

如果没有设置$GOROOT环境变量，在运行gomake时就必须使用第二种makefile。即使系统中的GNU Make的名字是gmake而不是make，Gomake也能正常的调用它。

TARG 是这个包的目标安装路径，就是客户用来导入这个包的字符串。在Go的源代码树种，这个字符串必须跟Makefile中的目录保持一致，不需 要$GOROOT/src/pkg/前缀。在Go的源代码树之外，则可以使用任何跟标准Go包名称不冲突的TARG。一个常见的规则是用一个独有的名称把 自己的包组合在一起，例如：myname/tree、 myname/filter等。注意，即使包的源代码是放在Go源代码树外部，为了便于编译器找到你的包，运行make install之后最好也把编译后的包放到标准位置，即$GOROOT/pkg。

GOFILES是创建包所需要编译的源代码文件清单。用反斜杠符号\就能将这份清单分成多行，方便排序。

如果在Go的源代码树中新建包目录，只需要将其添加到$GOROOT/src/pkg/Makefile的清单中，就能将其包含在标准构建中。然后运行：

```
cd $GOROOT/src/pkg
./deps.bash
```

这是更新依赖文件Make.deps。（每次运行all.bash或make.bash时都会自动执行此操作。）

如果是修改一个已有的包，就不需要编辑$GOROOT/src/pkg/Makefile，不过运行deps.bash还是必须的。

6.3.2. Go源文件

对于每个源代码文件，在Makefile中的命令首先是包的名称，该名称也是导入包的默认名称。（同一个包中所有文件必须使用同一个名称。）Go的规则是， 包的名称是导入路径的最后一个元素，例如以“crypto/rot13”为导入路径的包的名称应该是rot13。现在，Go工具还要求链接到同一个二进制 文件的所有包的名称都应该是唯一的，但这个限制很快就会被移除。

Go会一次性编译包中所有的源代码文件，所以源代码中可以试用其它文件中的常量、变量、类型和函数，而无需特别的安排或声明。

编写简洁易懂的Go代码超出了本文档的范围。Effective Go对此有介绍。

6.4. 测试

Go 有一个名为gotest的轻量级测试框架。编写测试首先要创建一个文件名以_test.go结尾的文件，然后在其中加入名为TestXXX且签名是(t *testing.T)的函数。测试框架会逐个地运行此类函数；如果函数调用了失败函数，例如t.Error或t.Fail，测试就会失败。gotest命令的文档和testing包的文档中有关于测试的详细信息。

不需要在Makefile中列出*_test.go文件。

运行make test或gotest就能运行测试（两个命令是等价的）。如果只需要运行单个测试文件中的测试，例如one_test.go，执行gotest one_test.go即可。

如果关心程序的性能，可以添加一个Benchmark函数（详见gotest命令文档），然后用gotest -benchmarks=.运行该函数。

代码通过测试后，就可以提交给别人审查了。

6.5. 一个带测试的演示包

这个演示用的包numbers含有一个函数Double，其参数为一个整数，返回值则是将该整数乘以2。这个包由三个文件组成：

第一个，包的实现，numbers.go:

```
package numbers

func Double(i int) int {
    return i * 2
}
```

接下来是测试，numbers_test.go：

```
package numbers

import (
    "testing"
)

type doubleTest struct {
    in, out int
}

var doubleTests = []doubleTest{
    doubleTest{1, 2},
    doubleTest{2, 4},
    doubleTest{-5, -10},
}

func TestDouble(t *testing.T) {
    for _, dt := range doubleTests {
        v := Double(dt.in)
        if v != dt.out {
            t.Errorf("Double(%d) = %d, want %d.", dt.in, v, dt.ou
        }
    }
}
```

最后是Makefile：

```
include $(GOROOT)/src/Make.inc
TARG=numbers
GOFILES=      numbers.go  include $(GOROOT)/src/Make.pkg
```

运行gomake构建并将这个包安装到$GOROOT/pkg/（可供系统上的所有程序使用）。

执行gotest测试会重建这个包，包括numbers_test.go文件，然后运行TextDouble函数。输出“PASS”表示所有测试都成功通过。把实现中的乘数从2改成3就能看到测试失败的报告。

更多细节请参考gotest文档和testing包的文档。

7. Codelab: 编写Web程序

7.1. 简介

这个例子涉及到的技术：

- 创建一个数据类型，含有load和save函数
- 基于http包创建web程序
- 基于template包的html模板技术
- 使用regexp包验证用户输入
- 使用闭包

假设读者有以下知识：

- 基本的编程经验
- web程序的基础技术（HTTP, HTML）
- UNIX 命令行

7.2. 开始

首先，要有一个Linux, OS X, or FreeBSD系统，可以运行go程序。如果没有的话，可以安装一个虚拟机(如VirtualBox)或者 Virtual Private Server。

安装Go环境：(请参考 Installation Instructions).

创建一个新的目录，并且进入该目录：

```
$ mkdir ~/gowiki
$ cd ~/gowiki
```

创建一个wiki.go文件，用你喜欢的编辑器打开，然后添加以下代码：

```
package main

import (
    "fmt"
    "io/ioutil"
    "os"
)
```

我们从go的标准库导入fmt, ioutil 和 os包。 以后，当实现其他功能时，我们会根据需要导入更多包。

7.3. 数据结构

我们先定义一个结构类型，用于保存数据。wiki系统由一组互联的wiki页面组成，每个wiki页面包含内容和标题。我们定义wiki页面为结构page， 如下：

```
type page struct {
    title    string
    body    []byte
}
```

类型[]byte表示一个byte slice。(参考Effective Go了解slices的更多信息) 成员body之所以定义为[]byte而不是string类型，是因为[]byte可以直接使用io包的功能。

结构体page描述了一个页面在内存中的存储方式。但是，如果要将数据保存到磁盘的话，还需要给page类型增加save方法：

```
func (p *page) save() os.Error {
    filename := p.title + ".txt"
    return ioutil.WriteFile(filename, p.body, 0600)
}
```

类型方法的签名可以这样解读：“save为page类型的方法，方法的调用者为page类型的指针变量p。该成员函数没有参数，返回值为os.Error，表示错误信息。”

该方法会将page结构的body部分保存到文本文件中。为了简单，我们用title作为文本文件的名字。

方法save的返回值类型为os.Error，对应WriteFile（标准库函数，将byte slice写到文件中）的返回值。通过返回os.Error值，可以判断发生错误的类型。如果没有错误，那么返回nil(指针、接口和其他一些类型的零值)。

WriteFile的第三个参数为八进制的0600，表示仅当前用户拥有新创建文件的读写权限。(参考Unix手册 open(2))

下面的函数加载一个页面：

```
func loadPage(title string) *page {
    filename := title + ".txt"
    body, _ := ioutil.ReadFile(filename)
    return &page{title: title, body: body}
}
```

函数loadPage根据页面标题从对应文件读取页面的内容，并且构造一个新的 page变量——对应一个页面。

go中函数（以及成员方法）可以返回多个值。标准库中的io.ReadFile在返回[]byte的同时还返回os.Error类型的错误信息。前面的代码中我们用下划线“_”丢弃了错误信息。

但是ReadFile可能会发生错误，例如请求的文件不存在。因此，我们给函数的返回值增加一个错误信息。

```
func loadPage(title string) (*page, os.Error) {
    filename := title + ".txt"
    body, err := ioutil.ReadFile(filename)
    if err != nil {
        return nil, err
    }
    return &page{title: title, body: body}, nil
}
```

现在调用者可以检测第二个返回值，如果为nil就表示成功装载页面。否则，调用者可以得到一个os.Error对象。（关于错误的更多信息可以参考os package documentation）

现在，我们有了一个简单的数据结构，可以保存到文件中，或者从文件加载。我们创建一个main函数，测试相关功能。

```
func main() {
    p1 := &page{title: "TestPage", body: []byte("This is a sample
    p1.save()
    p2, _ := loadPage("TestPage")
    fmt.Println(string(p2.body))
}
```

编译后运行以上程序的话，会创建一个TestPage.txt文件，用于保存p1对应的页面内容。然后，从文件读取页面内容到p2，并且将p2的值打印到 屏幕。

可以用类似以下命令编译运行程序：

```
$ 8g wiki.go
$ 8l wiki.8
$ ./8.out
This is a sample page.
```

(命令8g和8l对应GOARCH=386。如果是amd64系统，可以用6g和6l)

点击这里查看我们当前的代码。

7.4. 使用http包

下面是一个完整的web server例子：

```
package main

import (
    "fmt"
    "http"
)

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hi there, I love %s!", r.URL.Path[1:])
}

func main() {
    http.HandleFunc("/", handler)
    http.ListenAndServe(":8080", nil)
}
```

在main函数中，http.HandleFunc设置所有对根目录请求的处理函数为handler。

然后调用http.ListenAndServe，在8080端口开始监听（第二个参数暂时可以忽略）。然后程序将阻塞，直到退出。

函数handler为http.HandlerFunc类型，它包含http.Conn和http.Request两个类型的参数。

其中http.Conn对应服务器的http连接，我们可以通过它向客户端发送数据。

类型为http.Request的参数对应一个客户端请求。其中r.URL.Path 为请求的地址，它是一个string类型变量。我们用[1:]在Path上创建 一个slice，对应"/"之后的路径名。

启动该程序后，通过浏览器访问以下地址：

```
http://localhost:8080/monkeys
```

会看到以下输出内容：

```
Hi there, I love monkeys!
```

7.5. 基于http提供wiki页面

要使用http包，先将其导入：

```
import (
    "fmt"
    "http"
    "io/ioutil"
    "os"
)
```

然后创建一个用于浏览wiki的函数：

```
const lenPath = len("/view/")

func viewHandler(w http.ResponseWriter, r *http.Request) {
    title := r.URL.Path[lenPath:]
    p, _ := loadPage(title)
    fmt.Fprintf(w, "<h1>%s</h1><div>%s</div>", p.title, p.body)
}
```

首先，这个函数从r.URL.Path(请求URL的path部分)中解析页面标题。全局常量lenPath保存"/view/"的长度，它是请求路径的前缀部分。Path总是以"/view/"开头，去掉前面的6个字符就可以得到页面标题。

然后加载页面数据，格式化为简单的HTML字符串，写到c中，c是一个http.Conn类型的参数。

注意这里使用下划线“_”忽略loadPage的os.Error返回值。 这不是一种好的做法，此处是为了保持简单。我们将在后面考虑这个问题。

为了使用这个处理函数(handler)，我们创建一个main函数。它使用viewHandler初始化http，把所有以/view/开头的请求转发给viewHandler处理。

```
func main() {
    http.HandleFunc("/view/", viewHandler)
    http.ListenAndServe(":8080", nil)
}
```

点击这里查看我们当前的代码。

让我们创建一些页面数据（例如as test.txt），编译，运行。

```
$ echo "Hello world" > test.txt
$ 8g wiki.go
$ 8l wiki.8
$ ./8.out
```

当服务器运行的时候，访问http://localhost:8080/view/test将显示一个页面，标题为“test”，内容为“Hello world”。

7.6. 编辑页面

编辑功能是wiki不可缺少的。现在，我们创建两个新的处理函数(handler)：editHandler显示"edit page"表单(form)，saveHandler保存表单(form)中的数据。

首先，将他们添加到main()函数中：

```
func main() {
    http.HandleFunc("/view/", viewHandler)
    http.HandleFunc("/edit/", editHandler)
    http.HandleFunc("/save/", saveHandler)
    http.ListenAndServe(":8080", nil)
}
```

函数editHandler加载页面(或者，如果页面不存在，创建一个空page 结构)并且显示为一个HTML表单(form)。

```
func editHandler(w http.ResponseWriter, r *http.Request) {
    title := r.URL.Path[lenPath:]
    p, err := loadPage(title)
    if err != nil {
        p = &page{title: title}
    }
    fmt.Fprintf(w, "<h1>Editing %s</h1>"+
        "<form action=\"/save/%s\" method=\"POST\">"+
        "<textarea name=\"body\">%s</textarea><br>"+
        "<input type=\"submit\" value=\"Save\">"+
        "</form>",
        p.title, p.title, p.body)
}
```

这个函数能够工作，但是硬编码的HTML非常丑陋。当然，我们有更好的办法。

7.7. template包

template包是GO标准库的一个部分。我们使用template将HTML存放在一个单独的文件中，可以更改编辑页面的布局而不用修改相关的GO代码。

首先，我们必须将template添加到导入列表：

```
import (
    "http"
    "io/ioutil"
    "os"
    "template"
)
```

创建一个包含HTML表单的模板文件。打开一个名为edit.html的新文件，添加下面的行：

```
<h1>Editing {title}</h1>

<form action="/save/{title}" method="POST">
<div><textarea name="body" rows="20" cols="80">{body|html}</texta
<div><input type="submit" value="Save"></div>
</form>
```

修改editHandler，用模板替代硬编码的HTML。

```
func editHandler(w http.ResponseWriter, r *http.Request) {
    title := r.URL.Path[lenPath:]
    p, err := loadPage(title)
    if err != nil {
        p = &page{title: title}
    }
    t, _ := template.ParseFile("edit.html", nil)
    t.Execute(p, w)
}
```

函数template.ParseFile读取edit.html的内容，返回*template.Template类型的数据。

方法t.Execute用p.title和p.body的值替换模板中所有的{title}和{body}，并且把结果写到http.Conn。

注意，在上面的模板中我们使用{body|html}。|html部分请求模板引擎在输出body的值之前，先将它传到html格式化器(formatter)，转义HTML字符（比如用&aamp;gt;替换>）。 这样做，可以阻止用户数据破坏表单HTML。

既然我们删除了fmt.Sprintf语句，我们可以删除导入列表中的"fmt"。

使用模板技术，我们可以为viewHandler创建一个模板，命名为view.html。

```
<h1>{title}</h1>

<p>[<a href="/edit/{title}">edit</a>]</p>

<div>{body}</div>
```

修改viewHandler：

```
func viewHandler(w http.ResponseWriter, r *http.Request) {
    title := r.URL.Path[lenPath:]
    p, _ := loadPage(title)
    t, _ := template.ParseFile("view.html", nil)
    t.Execute(p, w)
}
```

注意，在两个处理函数(handler)中使用了几乎完全相同的模板处理代码，我们可以把模板处理代码写成一个单独的函数，以消除重复。

```
func viewHandler(w http.ResponseWriter, r *http.Request) {
    title := r.URL.Path[lenPath:]
    p, _ := loadPage(title)
    renderTemplate(w, "view", p)
}

func editHandler(w http.ResponseWriter, r *http.Request) {
    title := r.URL.Path[lenPath:]
    p, err := loadPage(title)
    if err != nil {
        p = &page{title: title}
    }
    renderTemplate(w, "edit", p)
}

func renderTemplate(w http.ResponseWriter, tmpl string, p *page)
    t, _ := template.ParseFile(tmpl+".html", nil)
    t.Execute(p, w)
}
```

现在，处理函数(handler)代码更短、更加简单。

7.8. 处理不存在的页面

当你访问/view/APageThatDoesntExist的时候会发生什么？程序将会崩溃。因为我们忽略了loadPage返回的错误。请求页不存在的时候，应该重定向客户端到编辑页，这样新的页面将会创建。

```
func viewHandler(w http.ResponseWriter, r *http.Request, title st
    p, err := loadPage(title)
    if err != nil {
        http.Redirect(w, r, "/edit/"+title, http.StatusFound)
        return
    }
    renderTemplate(w, "view", p)
}
```

函数http.Redirect添加HTTP状态码http.StatusFound (302)和报头Location到HTTP响应。

7.9. 储存页面

函数saveHandler处理表单提交。

```
func saveHandler(w http.ResponseWriter, r *http.Request) {
    title := r.URL.Path[lenPath:]
    body := r.FormValue("body")
    p := &page{title: title, body: []byte(body)}
    p.save()
    http.Redirect(w, r, "/view/"+title, http.StatusFound)
}
```

页面标题（在URL中）和表单中唯一的字段，body，储存在一个新的page中。然后调用save()方法将数据写到文件中，并且将客户重定向到/view/页面。

FormValue返回值的类型是string，在将它添加到page结构前，我们必须将其转换为[]byte类型。我们使用[]byte(body)执行转换。

7.10. 错误处理

在我们的程序中，有几个地方的错误被忽略了。这是一种很糟糕的方式，特别是在错误发生后，程序会崩溃。更好的方案是处理错误并返回错误消息给用户。这样做，当错误发生后，服务器可以继续运行，用户也会得到通知。

首先，我们处理renderTemplate中的错误：

```
func renderTemplate(w http.ResponseWriter, tmpl string, p *page)
    t, err := template.ParseFile(tmpl+".html", nil)
    if err != nil {
        http.Error(w, err.String(), http.StatusInternalServerErro
        return
    }
    err = t.Execute(p, w)
    if err != nil {
        http.Error(w, err.String(), http.StatusInternalServerErro
    }
}
```

函数http.Error发送一个特定的HTTP响应码（在这里表示“Internal Server Error”）和错误消息。

现在，让我们修复saveHandler：

```
func saveHandler(w http.ResponseWriter, r *http.Request, title st
    body := r.FormValue("body")
    p := &page{title: title, body: []byte(body)}
    err := p.save()
    if err != nil {
        http.Error(w, err.String(), http.StatusInternalServerErro
        return
    }
    http.Redirect(w, r, "/view/"+title, http.StatusFound)
}
```

p.save()中发生的任何错误都将报告给用户。

7.11. 模板缓存

代码中有一个低效率的地方：每次显示一个页面，renderTemplate都要调用ParseFile。更好的做法是在程序初始化的时候对每个模板调用ParseFile一次，将结果保存为*Template类型的值，在以后使用。

首先，我们创建一个全局map，命名为templates。templates用于储存*Template类型的值，使用string索引。

然后，我们创建一个init函数，init函数会在程序初始化的时候调用，在main函数之前。函数template.MustParseFile是ParseFile的一个封装，它不返回错误码，而是在错误发生的时候抛出(panic)一个错误。抛出错误(panic)在这里是合适的，如果模板不能加载，程序唯一能做的有意义的事就是退出。

```
func init() { for _, tmpl := range []string{"edit", "view"} { ter
```

使用带range语句的for循环访问一个常量数组中的每一个元素，这个常量数组中包含了我们想要加载的所有模板的名称。如果我们想要添加更多的模板，只要把模板名称添加的数组中就可以了。

修改renderTemplate函数，在templates中相应的Template上调用Execute方法：

```
func renderTemplate(w http.ResponseWriter, tmpl string, p *page)
    err := templates[tmpl].Execute(p, w)
    if err != nil {
        http.Error(w, err.String(), http.StatusInternalServerErr
    }
}
```

7.12. 验证

你可能已经发现，程序中有一个严重的安全漏洞：用户可以提供任意的路径在服务器上执行读写操作。为了消除这个问题，我们使用正则表达式验证页面的标题。

首先，添加"regexp"到导入列表。然后创建一个全局变量存储我们的验证正则表达式：

函数regexp.MustCompile解析并且编译正则表达式，返回一个regexp.Regexp对象。和template.MustParseFile类似，当表达式编译错误时，MustCompile抛出一个错误，而Compile在它的第二个返回参数中返回一个os.Error。

现在，我们编写一个函数，它从请求URL解析中解析页面标题，并且使用titleValidator进行验证：

```
func getTitle(w http.ResponseWriter, r *http.Request) (title stri
    title = r.URL.Path[lenPath:]
    if !titleValidator.MatchString(title) {
        http.NotFound(w, r)
        err = os.NewError("Invalid Page Title")
    }
    return
}
```

如果标题有效，它返回一个nil错误值。如果无效，它写"404 Not Found"错误到HTTP连接中，并且返回一个错误对象。

修改所有的处理函数，使用getTitle获取页面标题：

```
func viewHandler(w http.ResponseWriter, r *http.Request) {
    title, err := getTitle(w, r)
    if err != nil {
        return
    }
    p, err := loadPage(title)
    if err != nil {
        http.Redirect(w, r, "/edit/"+title, http.StatusFound)
        return
    }
    renderTemplate(w, "view", p)
}

func editHandler(w http.ResponseWriter, r *http.Request) {
    title, err := getTitle(w, r)
    if err != nil {
        return
    }
    p, err := loadPage(title)
    if err != nil {
        p = &page{title: title}
    }
    renderTemplate(w, "edit", p)
}

func saveHandler(w http.ResponseWriter, r *http.Request) {
    title, err := getTitle(w, r)
    if err != nil {
        return
    }
    body := r.FormValue("body")
    p := &page{title: title, body: []byte(body)}
    err = p.save()
    if err != nil {
        http.Error(w, err.String(), http.StatusInternalServerErr
        return
    }
    http.Redirect(w, r, "/view/"+title, http.StatusFound)
}
```

7.13. 函数文本和闭包

处理函数(handler)中捕捉错误是一些类似的重复代码。如果我们想将捕捉错误的代码封装成一个函数，应该怎么做？GO的函数文本提供了强大的抽象能力，可以帮我们做到这点。

首先，我们重写每个处理函数的定义，让它们接受标题字符串:

定义一个封装函数，接受上面定义的函数类型，返回http.HandlerFunc（可以传送给函数http.HandleFunc）。

```
func makeHandler(fn func (http.ResponseWriter, *http.Request, str
    return func(w http.ResponseWriter, r *http.Request) {
        // Here we will extract the page title from the Request,
        // and call the provided handler 'fn'
    }
}
```

返回的函数称为闭包，因为它包含了定义在它外面的值。在这里，变量fn（makeHandler的唯一参数）被闭包包含。fn是我们的处理函数，save、edit、或view。

我们可以把getTitle的代码复制到这里（有一些小的变动）：

```
func makeHandler(fn func(http.ResponseWriter, *http.Request, stri
    return func(w http.ResponseWriter, r *http.Request) {
        title := r.URL.Path[lenPath:]
        if !titleValidator.MatchString(title) {
            http.NotFound(w, r)
            return
        }
        fn(w, r, title)
    }
}
```

makeHandler返回的闭包是一个函数，它有两个参数，http.Conn和http.Request（因此，它是http.HandlerFunc）。闭包从请求路径解析title，使用titleValidator验证标题。如果title无效，使用函数http.NotFound将错误写到Conn。如果title有效，封装的处理函数fn将被调用，参数为Conn, Request, 和title。

在main函数中，我们用makeHandler封装所有处理函数：

```
func main() {
    http.HandleFunc("/view/", makeHandler(viewHandler))
    http.HandleFunc("/edit/", makeHandler(editHandler))
    http.HandleFunc("/save/", makeHandler(saveHandler))
    http.ListenAndServe(":8080", nil)
}
```

最后，我们可以删除处理函数中的getTitle，让处理函数更简单。

```
func viewHandler(w http.ResponseWriter, r *http.Request, title st
    p, err := loadPage(title)
    if err != nil {
        http.Redirect(w, r, "/edit/"+title, http.StatusFound)
        return
    }
    renderTemplate(w, "view", p)
}

func editHandler(w http.ResponseWriter, r *http.Request, title st
    p, err := loadPage(title)
    if err != nil {
        p = &page{title: title}
    }
    renderTemplate(w, "edit", p)
}

func saveHandler(w http.ResponseWriter, r *http.Request, title st
    body := r.FormValue("body")
    p := &page{title: title, body: []byte(body)}
    err := p.save()
    if err != nil {
        http.Error(w, err.String(), http.StatusInternalServerErro
        return
    }
    http.Redirect(w, r, "/view/"+title, http.StatusFound)
}
```

7.14. 试试！

点击这里查看最终的代码

重新编译代码，运行程序：

```
$ 8g wiki.go
$ 8l wiki.8
$ ./8.out
```

访问http://localhost:8080/view/ANewPage将会出现一个编辑表单。你可以输入一些文版，点击“Save”，重定向到新的页面。

7.15. 其他任务

这里有一些简单的任务，你可以自己解决：

- 把模板文件存放在tmpl/目录，页面数据存放在data/目录。
- 增加一个处理函数(handler)，将对根目录的请求重定向到/view/FrontPage。
- 修饰页面模板，使其成为有效的HTML文件。添加CSS规则。
- 实现页内链接。将[PageName]修改为PageName。(提示：可以使用regexp.ReplaceAllFunc达到这个效果)

8. 针对C++程序员指南

Go和C＋＋一样，也是一门系统编程语言。该文档主要面向有C＋＋经验的程序开发人员。 它讨论 了Go和C＋＋的不同之处，当然也讨论了一些相似之处。

如果是想要Go的概要介绍，请参考 Go tutorial 和 Effective Go。

关于语言细节的正式说明，请参考 Go spec。

8.1. 概念差异

- Go没有支持构造和析构的class类型，也没有继承和虚函数的概念。但是go提供接口interfaces 支持，我们可以把接口看作是C++中模板类似的技术。
- Go提供垃圾内存回收支持。我们没有必要显式释放内存，go的运行时系统会帮我们收集垃圾内存。
- Go中有指针，但是没有指针算术。因此，你不可能通过指针以字节方式来遍历一个字符串。 数组一个普通类型变量。当用数组作为参数调用函数时，将会复制整个数组。当然，Go语言中一般用切片 （slices）代替数组作为参数，切片是建立在底层数组地址之上的，因此传递的是数组的 地址。切片在后面会详细讨论。
- 内建对字符串的支持。并且字符串创建后就不能修改。
- 内建hash表支持，术语叫字典（map）。
- 语言本身提供并发和管道通讯功能，细节在后面会讨论。
- 有少数类型是通过引用传递（字典和管道，将在后面讨论）。也就是说，将字典传递给一个 函数不会复制整个字典，而且函数对字典的修改会影响到函数调用者的字典数据。这和C++中引用 概念类似。
- Go不使用头文件。每个源文件都被定义在特定的包package中，在包中以大写字母定义的对象（例如类型，常量，变量，函数等）对外是可见的，可以被别的代码导入使用。
- Go不会作隐式类型转换。如果在不同类型之间赋值，必须强制转换类型。
- Go不支持函数重载，也不支持运算符定义。
- Go不支持const和volatile 修饰符。
- Go使用nil表示无效的指针，C++中使用NULL或0 表示空指针。

8.2. 语法

Go中变量的声明语法和C++相反。定义变量时，先写变量的名字，然后是变量的类型。这样 不会出现像C++中， 类型不能匹配后面所有变量的情况（指针类型）。而且语法清晰，便于 阅读。

```
Go                           C++
var v1 int                // int v1;
var v2 string             // const std::string v2;  (approximatel
var v3 [10]int            // int v3[10];
var v4 []int              // int* v4;  (approximately 近似等价)
var v5 struct { f int }   // struct { int f; } v5;
var v6 *int               // int* v6;  (but no pointer arithmetic
var v7 map[string]int     // unordered_map<string, int>* v7;  (ap
var v8 func(a int) int    // int (*v8)(int a);
```

变量的声明通常是从某些关键字开始，例如var， func，const或type。对于类型的专有方法定义， 在变量名前面还要加上对应该方法发类型对象变量，细节清参考discussion of interfaces。

你也可以在关键字后面加括号，这样可以同时定义多个变量。

```
var (
    i int
    m float
)
```

When declaring a function, you must either provide a name for each parameter or not provide a name for any parameter; you can't omit some names and provide others. You may group several names with the same type:

定义函数的时候，你可以指定每个参数的名字或者不指定任何参数名字，但是你不能只指定部分函数参数的 名字。如果是相邻的参数是相同的类型，也可以统一指定类型。

```
func f(i, j, k int, s, t string)
```

对于变量，可以在定时进行初始化。对于这种情况，我们可以省略变量的类型部分，因为Go编译器 可以根据初始化的值推导出变量的类型。

```
var v = *p
```

如果变量定义时没有初始化，则必须指定类型。没有显式初始化的变量，会被自动初始化为空的值， 例如0，nil等。Go不存在完全未初始化的变量。

用:=操作符，还有更简短的定义语法：

```
v1 := v2
```

和下面语句等价：

```
var v1 = v2
```

Go还提供多个变量同时赋值：

```
i, j = j, i    // Swap i and j.
```

函数也可以返回多个值，多个返回值需要用括号括起来。返回值可以用一个等于符号赋给 多个变量。

```
func f() (i int, j int) { ... }
v1, v2 = f()
```

Go中使用很少的分号，虽然每个语句之间实际上是用分号分割的。因为，go编译器会在看似 完整的语句末尾自动添加分号（具体细节清参考Go语言手册）。 当然，自动添加分号也可能带来一些问题。例如：

```
func g()
{                  // INVALID
}
```

在g()函数后面会被自动添加分号，导致函数编译出错。下面的代码也有类似的 问题：

```
if x {
}
else {             // INVALID
}
```

在第一个花括号}的后面会被自动添加分号，导致else语句 出现语法错误。

分号可以用来分割语句，你仍然可以安装C++的方式来使用分号。不过Go语言中，常常省略不 必要的分号。 只有在 循环语句的初始化部分，或者一行写多个语句的时候才是必须的。

继续前面的问题。我们并不用担心因为花括号的位置导致的编译错误，因此我们可以用 gofmt 来排版程序代码。 gofmt 工具总是可以将代码排版成统一的风格。While the style may initially seem odd, it is as good as any other style, and familiarity will lead to comfort.

当用指针访问结构体的时候，我们用.代替->语法。 因此，用结构体类型和结构体指针类型访问结构体成员的语法是一样的。

```
type myStruct struct { i int }
var v9 myStruct              // v9 has structure type
var p9 *myStruct             // p9 is a pointer to a structure
f(v9.i, p9.i)
```

Go不要求在if语句的条件部分用小括弧，但是要求if后面的代码 部分必须有花括弧。类似的规则也适用于for和switch等语句。

```
if a < b { f() }             // Valid
if (a < b) { f() }           // Valid (condition is a parenthesiz
if (a < b) f()               // INVALID
for i = 0; i < 10; i++ {}    // Valid
for (i = 0; i < 10; i++) {}  // INVALID
```

Go语言中没有while和do/while循环语句。我们可以用只有一个 条件语句的for来代替while循环。如果省略for 的条件部分，则是一个无限循环。

Go增加了带标号的break 和continue语法。不过标号必须 是针对for，switch或select代码段的。

对于switch语句，case匹配后不会再继续匹配后续的部分。 对于没有任何匹配的情况，可以用fallthrough 语句。

```
switch i {
case 0:  // empty case body
case 1:
    f()  // f is not called when i == 0!
}
```

case语句还可以带多个值：

```
switch i {
case 0, 1:
    f()  // f is called if i == 0 || i == 1.
}
```

case语句不一定必须是整数或整数常量。如果省略switch的 要匹配的值，那么case可以是任意的条件语言。

```
switch {
case i < 0:
    f1()
case i == 0:
    f2()
case i > 0:
    f3()
}
```

++ 和 -- 不再是表达式，它们只能在语句中使用。因此， c = *p*++ 是错误的。语句 p++ 的含义也完全不同，在go中等价于 (*p)++ 。

defer可以用于指定函数返回前要执行的语句。

```
fd := open("filename")
defer close(fd)         // fd will be closed when this function r
```

8.3. 常量

Go语言中的常量可以没有固定类型（untyped）。我们可以用const和一个 untyped 类型的初始值来 定义untyped常量。如果是untyped常量， 那么常量在使用的时候会根据上下文自动进行隐含的类型转换。 这样，可以更自由的使用untyped常量。

```
var a uint
f(a + 1)  // untyped numeric constant "1" becomes typed as uint
```

untyped类型常量的大小也没有限制。只有在最终使用的地方才有大小的限制。

```
const huge = 1 << 100
f(huge >> 98)
```

Go没有枚举类型（enums）。作为代替，可以在一个独立的const区域中使用 iota 来生成递增的值。如果const中，常量没有初始值则会 用前面的初始化表达式代替。

```
const (
    red = iota   // red == 0
    blue         // blue == 1
    green        // green == 2
)
```

8.4. Slices(切片)

切片（slice）底层对应类结构体，主要包含以下信息：指向数据的指针，有效数据的数目，和总 的内存空间大小。 切片支持用语法获取底层数组的某个元素。内建的len 方法可以获取切片的长度。内建的cap返回切片的最大容量。

对于一个数组或另一个切片，我们用aI:J?语句再它基础上创建一个新的切片。 这个新创建的切片底层 引用a（数组或之前的另一个切片），从数组的I位置 开始，到数组的J位置结束。新切片的长度是J - I。 新切片的 容量是数组的容量减去切片在数组中的开始位置I。我们还可以将数组的地址直接赋给 切片：s = &a， 这默认是对应整个数组，和这个语句等价：s = a0:len(a)?。

因此，我们在在C++中使用指针的地方用切片来代替。例如，创建一个100?byte类型的 值（100个字节的数组， 或许是做为缓冲用）。但是，在将数组传递给函数的时候不想复制整个数组（go语言 中数组是值，函数参数 传值是复制的），可以将函数参数定一个为byte切片类型， 从而实现只传递数组地址的目的。不过我们 并不需要像C++中那样传递缓存的长度——在Go中它们已经包含 在切片信息中了。

切片还可以应用于字符串。当需要将某个字符串的字串作为你新字符产返回的时候可以用切片代替。 因为go中的字符串是不可修改的，因此使用字符串切片并不需要分配新的内存空间。

8.5. 构造值对象

Go有一个内建的new函数，用于在堆上为任意类型变量分配一个空间。新分配的内存会内自动初始化为0。 例如，new(int) 会在堆上分配一个整型大小的空间， 然后初始化为0，然后返回 *int 类型的地址。 和C++中不同的 是，new是一个函数而不是运算符，因此 new int 用法是错误的。

对于字典和管道，必须用内建的make函数分配空间。对于没有初始化的字典或 管道变量，会被自动初始化为nil。 调用make(mapint?int) 返回一个新的字典空间， 类型为mapint?int。需要强调的是，make 返回的是值， 而不是指针！与此对应的是，字典和管道是通过引用传递的。对于make 分配字典空间， 还可以有一个可选的函数， 用于指定字典的容量。如果是用于创建管道，则可选的参数 对应管道的缓冲大小，默认0表示不缓存。

make函数还可以用于创建切片。这时，会在堆中分配一个不可见的数组，然后返回 对这个数组引用的切片。 对于切片，make函数除了一个指定切片大小的参数外， 还有一个可选的用于指定切片容量的参数（最多有3个参数）。 例如，make(int, 10, 20)， 用于创建一个大小是10，容量是20的切片。当然，用new函数也能实现： new(20?int)0:10?。go支持垃圾内存自动回收，因此新分配的内存空间没有 任何切片引用的时候，可能会被自动释放。

8.6. Interfaces(接口)

C++提供了class，类继承和模板，类似的go语言提供了接口支持。go中的接口和C++中的纯虚 基类 （只有虚函数，没有数据成员）类似。在Go语言中，任何实现了接口的函数的类型，都可以 看作是接口的一个实现。 类型在实现某个接口的时候，不需要显式关联该接口的信息。接口的实现 和接口的定义完全分离了。

类型的方法和普通函数定义类似，只是前面多了一个对象接收者receiver。 对象接受者和C++中的this指针类似。

```
type myType struct { i int }
func (p *myType) get() int { return p.i }
```

方法get依附于myType类型。myType对象在 函数中对应p。

方法在命名的类型上定义。如果，改变类型的话，那么就是针对新类型的另一个函数了。

如果要在内建类型上定义方法，则需要给内建类型重新指定一个名字，然后在新指定名字的类型上 定义方法。新定义的类型和内建的类型是有区别的。

```
type myInteger int
func (p myInteger) get() int { return int(p) } // Conversion requ
func f(i int) { }
var v myInteger
// f(v) is invalid.
// f(int(v)) is valid; int(v) has no defined methods.
```

把方法抽象到接口：

```
type myInterface interface {
        get() int
        set(i int)
}
```

为了让我们前面定义的myType满足接口，需要再增加一个方法：

```
func (p *myType) set(i int) { p.i = i }
```

现在，任何以myInterface类型作为参数的函数，都可以用 *myType 类型传递了。

```
func getAndSet(x myInterface) {}
func f1() {
        var p myType
        getAndSet(&p)
}
```

以C++的观点来看，如果把myInterface看作一个纯虚基类，那么实现了 set 和 get 方法的 *myType 自动成为 了从myInterface纯虚基类继承的子类了。在Go中，一个类型可以同时 实现多种接口。

使用匿名成员，我们可以模拟C++中类的继承机制。

```
type myChildType struct { myType; j int }
func (p *myChildType) get() int { p.j++; return p.myType.get() }
```

这里的myChildType可以看作是myType的子类。

```
func f2() {
        var p myChildType
        getAndSet(&p)
}
```

myChildType类型中是有set方法的。在go中，匿名成员的方法 会默认被提升为类型本身的方法。 因为myChildType含有一个myType 类型的匿名成员，因此也就继承了myType中的set方法， 另一个 get方法则相当于被重载了。

不过这和C++也不是完全等价的。当一个匿名方法被调用的时候，方法对应的类型对象 是匿名成员类型， 并不是当前类型！换言之，匿名成员上的方法并不是C++中的虚函数。 如果你需要模拟虚函数机制， 那么可以使用接口。

一个接口类型的变量可以通过接口的一个内建的特殊方法转换为另一个接口类型变量。 这是由运行时库动态完成的， 和C++中的dynamic_cast有些类似。 但是在Go语言中，两个相互转换的接口类型之间并不需要什么信息。

```
type myPrintInterface interface {
  print()
}
func f3(x myInterface) {
        x.(myPrintInterface).print()  // type assertion to myPrin
}
```

向myPrintInterface类型的转换是动态的。它可以工作在底层实现了 print方法的变量上。

因为，这里动态类型转换机制，我们可以用它来模拟实现C++中的模板功能。这里我们需要 定一个最小的接口：

```
type Any interface { }
```

该接口可以持有任意类型的数据，但是在使用的时候需要将该接口变量转换为需要的类型。 因为，这里类型转换是动态实现的，因此，没有办法定义像C++中的内联函数。类型的验证 由运行时库完成， 我们可以调用该变量类型支持的所有方法。

```
type iterator interface {
        get() Any
        set(v Any)
        increment()
        equal(arg *iterator) bool
}
```

8.7. Goroutines

Go语言中使用go可以启动一个goroutine。goroutine 和线程的概念类似，和程序共享一个地址空间。

goroutines和支持多路并发草组系统中的协程（coroutines）类似，用户不用关心具体 的实现细节。

```
func server(i int) {
    for {
        print(i)
        sys.sleep(10)
    }
}
go server(1)
go server(2)
```

（需要注意的是server函数中的for循环语句和 C++ while (true)的循环类似。）

Goroutines资源开销小，比较廉价。

go也可以用于启动新定义的内部函数（闭包）为Goroutines。

```
var g int
go func(i int) {
        s := 0
        for j := 0; j < i; j++ { s += j }
        g = s
}(1000)  // Passes argument 1000 to the function literal.
```

8.8. Channels(管道)

管道可以用于两个goroutines之间的通讯。我们可以用管道传递任意类脂的变量。Go语言中管道是 廉价并且便捷的。 二元操作符 <- 用于向管道发送数据。一元操作符<- 用于从管道接收数据。在函数参数中，管道通过引用传递给函数。

虽然go语言的标准库中提供了互斥的支持，但是我们也可以用一个单一的goroutine提供对变量的 共享操作。 例如，下面的函数用于管理对变量的读写操作。

```
type cmd struct { get bool; val int }
func manager(ch chan cmd) {
        var val int = 0
        for {
                c := <- ch
                if c.get { c.val = val ch <- c }
                else { val = c.val }
        }
}
```

在这个例子中，管道被同时用于输入和输出。但是当多个goroutines对变量操作时可能导致 问题：对管道的读操作可能读到的是请求命令。解决的方法是将命令和数据分为不同的管道。

```
type cmd2 struct { get bool; val int; ch <- chan int }
func manager2(ch chan cmd2) {
        var val int = 0
        for {
                c := <- ch
                if c.get { c.ch <- val }
                else { val = c.val }
        }
}
```

这里掩饰了如何使用manager2：

```
func f4(ch <- chan cmd2) int {
        myCh := make(chan int)
        c := cmd2{ true, 0, myCh }   // Composite literal syntax
        ch <- c
        return <-myCh
}
```

9. 内存模型

9.1. 简介

Go的你内存模型可以保证对一个变量的读操作可以侦测到令一个 goroutine 中对给变量进行的写操作。

9.2. Happens Before

在一个单独的goroutine中，变量的读和写操作顺序和代码所写的顺序一致。因此，在变量值没有 被改变的时候，编译器和处理器可能会记录变量的操作顺序。但是，这种先验性的顺序记录会导致 在两个不同的goroutine对变量操作 顺序记录有差别。例如，一个goroutine执行 a = 1; b = 2;， 但是在另一个goroutine中可能会现感知到b被更新。

为了解决这种二义性问题，Go语言中引进一个happens before的概念，它用于描述对内存操作的先后顺序问题。如果 事件 e1 happens before 事件 e2，我们说 事件 e2 happens after e1。 如果，

```
事件e1 does not happen before 事件 e2，并且 does not happen after e2，
```

。

对于一个单一的goroutine，happens before 的顺序和代码的顺序是一致的。

如果能满足以下的条件，一个对变量v的读事件r可以 感知到另一个对变量v的写事件w：

1. 写事件w happens before 读事件r。
2. 没有既满足 happens after w 同时满主 happens before r的对变量v的写事件w。

为了保证读事件r可以感知对变量v的写事件，我们首先要 确保w是变量v的唯一的写事件。同时还要满足以下条件：

1. 写事件w happens before 读事件r。
2. 其他对变量v的访问必须 happens before 写事件w 或者 happens after 读事件r。

第二组条件比第一组条件更加严格。因为，它要求在w和 r并行执行的程序中不能再有其他的读操作。

对于在单一的goroutine中两组条件是等价的，读事件可以确保感知到对变量的写事件。但是，对于在 两个goroutines共享变量v，我们必须通过同步事件来保证 happens-before 条件 （这是读事件感知写事件的必要条件）。

将变量v自动初始化为零也是属于这个内存操作模型。

读写超过一个机器字长度的数据，顺序也是不能保证的。

9.3. 同步(Synchronization)

9.3.1. 初始化

程序的初始化在一个独立的goroutine中执行。在初始化过程中创建的goroutine将在第一个用于初始化goroutine执行完成后启动。

如果包p导入了包q，包q的init 初始化函数将在包p的初始化之前执行。

程序的入口函数 main.main 则是在所有的 init 函数执行完成 之后启动。

在任意init函数中新创建的goroutines，将在所有的init 函数完成后执行。

9.3.2. Goroutine的创建

用于启动goroutine的go语句在goroutine之前运行。

例如，下面的程序：

```
var a string;

func f() {
        print(a);
}

func hello() {
        a = "hello, world";
        go f();
}
```

调用hello函数，会在某个时刻打印“hello, world”（有可能是在hello函数返回之后）。

9.3.3. Channel communication 管道通信

用管道通信是两个goroutines之间同步的主要方法。在管道上执行的发送操作会关联到该管道的 接收操作，这通常对应goroutines。

管道上的发送操作发生在管道的接收完成之前（happens before）。

例如这个程序：

```
var c = make(chan int, 10)
var a string

func f() {
        a = "hello, world";
        c <- 0;
}

func main() {
        go f();
        <-c;
        print(a);
}
```

可以确保会输出"hello, world"。因为，a的赋值发生在向管道 c发送数据之前，而管道的发送操作在管道接收完成之前发生。 因此，在print 的时候，a已经被赋值。

从一个unbuffered管道接收数据在向管道发送数据完成之前发送。

下面的是示例程序：

```
var c = make(chan int)
var a string

func f() {
        a = "hello, world";
        <-c;
}
func main() {
        go f();
        c <- 0;
        print(a);
}
```

同样可以确保输出“hello, world”。因为，a的赋值在从管道接收数据 前发生，而从管道接收数据操作在向unbuffered 管道发送完成之前发生。所以，在print 的时候，a已经被赋值。

如果用的是缓冲管道（如 c = make(chan int, 1) ），将不能保证输出 “hello, world”结果（可能会是空字符串， 但肯定不会是他未知的字符串， 或导致程序崩溃）。

9.3.4. 锁

包sync实现了两种类型的锁： sync.Mutex 和 sync.RWMutex。

对于任意 sync.Mutex 或 sync.RWMutex 变量l。 如果 n < m ， 那么第n次 l.Unlock() 调用在第 m次 l.Lock() 调用返回前发生。

例如程序：

```
var l sync.Mutex
var a string

func f() {
        a = "hello, world";
        l.Unlock();
}

func main() {
        l.Lock();
        go f();
        l.Lock();
        print(a);
}
```

可以确保输出“hello, world”结果。因为，第一次 l.Unlock() 调用（在f函数中）在第二次 l.Lock() 调用 （在main 函数中）返回之前发生，也就是在 print 函数调用之前发生。

For any call to l.RLock on a sync.RWMutex variable l, there is an n such that the l.RLock happens (returns) after the n'th call to l.Unlock and the matching l.RUnlock happens before the n+1'th call to l.Lock.

9.3.5. Once

包once提供了一个在多个goroutines中进行初始化的方法。多个goroutines可以 通过 once.Do(f) 方式调用f函数。 但是，f函数 只会被执行一次，其他的调用将被阻塞直到唯一执行的f()返回。

once.Do(f) 中唯一执行的f()发生在所有的 once.Do(f) 返回之前。

有代码：

```
var a string

func setup() {
        a = "hello, world";
}

func doprint() {
        once.Do(setup);
        print(a);
}

func twoprint() {
        go doprint();
        go doprint();
}
```

调用twoprint会输出“hello, world”两次。第一次twoprint 函数会运行setup唯一一次。

9.4. 错误的同步方式

注意：变量读操作虽然可以侦测到变量的写操作，但是并不能保证对变量的读操作就一定发生在写操作之后。

例如：

```
var a, b int

func f() {
        a = 1;
        b = 2;
}

func g() {
        print(b);
        print(a);
}

func main() {
        go f();
        g();
}
```

函数g可能输出2，也可能输出0。

这种情形使得我们必须回避一些看似合理的用法。

这里用重复检测的方法来代替同步。在例子中，twoprint函数可能得到错误的值：

```
var a string
var done bool

func setup() {
        a = "hello, world";
        done = true;
}

func doprint() {
        if !done {
                once.Do(setup);
        }
        print(a);
}

func twoprint() {
        go doprint();
        go doprint();
}
```

在doprint函数中，写done暗示已经给a赋值了。 但是没有办法给出保证，函数可能输出空的值（在2个goroutines中同时执行到测试语句）。

另一个错误陷阱是忙等待：

```
var a string
var done bool

func setup() {
        a = "hello, world";
        done = true;
}

func main() {
        go setup();
        for !done {
        }
        print(a);
}
```

我们没有办法保证在main中看到了done值被修改的同时也 能看到a被修改，因此程序可能输出空字符串。 更坏的结果是，main 函数可能永远不知道done被修改，因为在两个线程之间没有同步操作，这样main 函数永远不能返回。

下面的用法本质上也是同样的问题.

```
type T struct {
        msg string;
}

var g *T

func setup() {
        t := new(T);
        t.msg = "hello, world";
        g = t;
}

func main() {
        go setup();
        for g == nil {
        }
        print(g.msg);
}
```

即使main观察到了 g != nil 条件并且退出了循环，但是任何然 不能保证它看到了g.msg的初始化之后的结果。

在这些例子中，只有一种解决方法：用显示的同步。

10. 附录

10.1. 命令行工具

```
Name          Synopsis
..
5a          5a is a version of the Plan 9 assembler.
5c          5c is a version of the Plan 9 C compiler.
5g          5g is the version of the gc compiler for the ARM. The S
5l          5l is a modified version of the Plan 9 linker.
6a          6a is a version of the Plan 9 assembler.
6c          6c is a version of the Plan 9 C compiler.
6g          6g is the version of the gc compiler for the x86-64.
6l          6l is a modified version of the Plan 9 linker.
8a          8a is a version of the Plan 9 assembler.
8c          8c is a version of the Plan 9 C compiler.
8g          8g is the version of the gc compiler for the x86.
8l          8l is a modified version of the Plan 9 linker.
cc          This directory contains the portable section of the Pla
cgo          Cgo enables the creation of Go packages that call C co
cov          Cov is a rudimentary code coverage tool.
ebnflint      Ebnflint verifies that EBNF productions are consisten
gc          Gc is the generic label for the family of Go compilers
godefs      Godefs is a bootstrapping tool for porting the Go runti
godoc         Godoc extracts and generates documentation for Go pr
gofmt         Gofmt Go程序格式化.
goinstall     Goinstall 尝试自动安装包的工具。
gomake      gomake是针对go语言对GNU make命令的简单包装。
gopack      Gopack is a variant of the Plan 9 ar tool.
gotest      Gotest is an automated testing tool for Go packages.
goyacc      Goyacc is a version of yacc for Go.
hgpatch      Hgpatch applies a patch to the local Mercurial reposit
ld          This directory contains the portable section of the Pla
nm          Nm is a version of the Plan 9 nm command.
prof          Prof is a rudimentary real-time profiler.
```

10.1.1. 8g

```
8g is the version of the gc compiler for the x86.
The $GOARCH for these tools is 386.

It reads .go files and outputs .8 files. The flags are documentec

There is no instruction optimizer, so the -N flag is a no-op.

8g是x86系统的gc编译器。

当$GOARCH设置为386时，该命令有效。输入“.go”文件，输出“.8”文件。命令行选择

该版本编译器没有进行指令优化，因此不支持``-N``参数。

用法: 8g [flags] file.go...

选项:
  -I DIR search for packages in DIR 指定包的查找路径
  -d print declarations 打印声明
  -e no limit on number of errors printed 打印全部的错误
  -f print stack frame structure 打印栈帧结构
  -h panic on an error 遇到错误停止
  -o file specify output file 指定输出文件名
  -S print the assembly language 打印编译后的汇编代码
  -V print the compiler version 打印编译器版本
  -u disable package unsafe 禁用unsafe包
  -w print the parse tree after typing 打印分析树
  -x print lex tokens 打印词法分析结果
```

10.1.2. 8l

```
8l is a modified version of the Plan 9 linker.  The original is d

8l是Plan 9系统连接器的修改版。文档在：

    http://plan9.bell-labs.com/magic/man2html/1/2l

Its target architecture is the x86, referred to by these tools fo
It reads files in .8 format generated by 8g, 8c, and 8a and emits
a binary called 8.out by default.

输出的目标文件针对x86系统（由于历史原因，这些工具中叫386）。
它的输入是8g, 8c,和8a生成的“.8”格式文件，然后默认输出8.out文件。

Major changes include:
    - support for ELF and Mach-O binary files
    - support for segmented stacks (this feature is implemented h

重要的改动：
    - 支持ELF和Mach-O格式的二进制文件
    - 支持分段的栈（该特性不是在编译器实现，是在这里）

Original options are listed in the link above.

原有的选项在上面提到的文档中。

Options new in this version:

这里是新加选项：

-d
    Elide the dynamic linking header.  With this option, the bina
    is statically linked and does not refer to dynld.  Without th
    (the default), the binary's contents are identical but it is
-H6
    Write Apple Mach-O binaries (default when $GOOS is darwin)
    生成Apple Mach-O格式文件（$GOOS为darwin）
-H7
    Write Linux ELF binaries (default when $GOOS is linux)
    生成Linux的ELF格式文件
-L dir1,dir2,..
    Search for libraries (package files) in the comma-separated l
    The default is the single location $GOROOT/pkg/$GOOS_386.
    包的目录列表，以逗号分隔。默认有一个目录$GOROOT/pkg/$GOOS_386.
-r dir1:dir2:...
    Set the dynamic linker search path when using ELF.
    设置动态链接的查找路径（针对ELF）
-V
    Print the linker version.
    输出连接器的版本号
```

10.1.3. 8a

```
8a is a version of the Plan 9 assembler.  The original is documer

8a是Plan 9的汇编器，文档在

    http://plan9.bell-labs.com/magic/man2html/1/2a

Its target architecture is the x86, referred to by these tools fc

目标是x86结构，由于历史原因，这些工具中叫386。
```

10.1.4. gomake

```
The gomake command runs GNU make with an appropriate environment
for using the conventional Go makefiles.  If $GOROOT is already
set in the environment, running gomake is exactly the same
as running make (or, on BSD systems, running gmake).

gomake是针对go语言对GNU make命令的简单包装。

如果已经设置了$GOROOT环境变量的话，gomake是和gmake等价的。如果没有设置$GO

用法: gomake [ 目标 ... ]

支持的目标:

    all (默认)
        build the package or command, but do not install it.
        构建全部的包和命令，但是不进行安装操作。

    install
        build and install the package or command
        构建全部的包和命令，然后安装。

    test
        run the tests (packages only)
        运行包的测试代码。

    bench
        run benchmarks (packages only)
        运行包的基准测试。

    clean
        remove object files from the current directory
        清空构建时生成的临时文件。

    nuke
        make clean and remove the installed package or command
        清空构建时生成的临时文件，并且删除已经安装的包和命令。

查看 http://golang.org/doc/code.html 页面，获取关于makefiles的详细信息
```

10.1.5. cgo

注：该命令有较大更新，有些特性提供的例子中没有展示。

```
Cgo enables the creation of Go packages that call C code.

cgo用于创建要调用C语言函数的包。

Usage: cgo [compiler options] file.go
```

```
The compiler options are passed through uninterpreted when
invoking gcc to compile the C parts of the package.

编译器的选项在调用gcc编译C代码的时候，传入gcc。

The input file.go is a syntactically valid Go source file that in
the pseudo-package "C" and then refers to types such as C.size_t,
variables such as C.stdout, or functions such as C.putchar.

file.go输入文件是一个导入了"C"虚拟包的go源文件。然后可以通过“C.”前缀访问C
的内容，如 C.size_t、C.stdout、C.putchar。

If the import of "C" is immediately preceded by a comment, that
comment is used as a header when compiling the C parts of
the package.  For example:

如果注释后紧跟着导入了"C"包，那么"C"之前的注释将作为有效的C代码处理。例如：

    // #include <stdio.h>
    // #include <errno.h>
    import "C"

C identifiers or field names that are keywords in Go can be
accessed by prefixing them with an underscore: if x points at
a C struct with a field named "type", x._type accesses the field.

The standard C numeric types are available under the names
C.char, C.schar (signed char), C.uchar (unsigned char),
C.short, C.ushort (unsigned short), C.int, C.uint (unsigned int),
C.long, C.ulong (unsigned long), C.longlong (long long),
C.ulonglong (unsigned long long), C.float, C.double.

标准的C数值类型：
C.char, C.schar (signed char), C.uchar (unsigned char),
C.short, C.ushort (unsigned short), C.int, C.uint (unsigned int),
C.long, C.ulong (unsigned long), C.longlong (long long),
C.ulonglong (unsigned long long), C.float, C.double.

To access a struct, union, or enum type directly, prefix it with
struct_, union_, or enum_, as in C.struct_stat.

如果是struct, union,  或 enum  类型，添加前缀struct_, union_, or enum
例如： C.struct_stat.

Any C function that returns a value may be called in a multiple
assignment context to retrieve both the return value and the
C errno variable as an os.Error.  For example:

任意有返回值的C函数，可以在go中当作返回多个值处理——第二个返回值转为os.Erro
例如：

    n, err := C.atoi("abc")
```

```
In C, a function argument written as a fixed size array
actually requires a pointer to the first element of the array.
C compilers are aware of this calling convention and adjust
the call accordingly, but Go cannot.  In Go, you must pass
the pointer to the first element explicitly: C.f(&x[0]).

C函数参数中，数组类型参数实际上是指向数组第一个元素的指针。在C语言中
可以直接将数组传递给函数参数，但是go不允许。go中必须明确传递第一个元素的
指针：C.f(&x[0]).

Cgo transforms the input file into four output files: two Go sour
files, a C file for 6c (or 8c or 5c), and a C file for gcc.

cgo处理后，每个输出的文件生成4个输出文件：2个是go文件，1个针对8c/6c的C文件，
1个针对gcc的C文件。

The standard package makefile rules in Make.pkg automate the
process of using cgo.  See $GOROOT/misc/cgo/stdio and
$GOROOT/misc/cgo/gmp for examples.

标准库的makefile模板文件Make.pkg支持cgo语法。例子请参考：$GOROOT/misc/c
和$GOROOT/misc/cgo/gmp。

Cgo does not yet work with gccgo.

cgo目前不支持gccgo。
```

10.1.6. gotest

```
Gotest is an automated testing tool for Go packages.

gotest是包的自动测试工具。

Normally a Go package is compiled without its test files.  Gotest
is a simple script that recompiles the package along with any fil
named *_test.go.  Functions in the test sources named TestXXX
(where XXX is any alphanumeric string starting with an upper case
letter) will be run when the binary is executed.  Gotest requires
that the package have a standard package Makefile, one that
includes go/src/Make.pkg.

包的测试文件默认是没有编译的。gotest是一个用于编译*_test.go文件的脚本。
测试文件中所有的TestXXX（XXX是大写字母开头的单词）函数会被执行。Gotest
需要包的makefile文件包含go/src/Make.pkg模板。

The test functions are run in the order they appear in the source
They should have signature

测试函数安装在源文件中出现的顺序被执行，必须是以下类型：
```

```
    func TestXXX(t *testing.T) { ... }

Benchmark functions can be written as well; they will be run only
when the -benchmarks flag is provided.  Benchmarks should have
signature

基准测试也已经支持，只要在命令行增加 -benchmarks 选项。基准测试函数的类型：

    func BenchmarkXXX(b *testing.B) { ... }

See the documentation of the testing package for more informatior

查看testing包文档，可以获取详细信息。

By default, gotest needs no arguments.  It compiles all the .go f
in the directory, including tests, and runs the tests.  If file r
are given, only those test files are added to the package.
(The non-test files are always compiled.)

gotest默认不需要参数。它编译目录中的所有go文件，包含测试文件，然后执行
测试。如果设置文件名参数，那么只有对应测试文件才会被编译执行（无测试的文件依然

The package is built in a special subdirectory so it does not
interfere with the non-test installation.

包构建的中间文件默认放在一个特殊的子目录，因此不会干扰测试。

Usage:
    gotest [pkg_test.go ...]

The resulting binary, called (for amd64) 6.out, has a couple of
arguments.

输出6.out文件（针对amd64）。有一组命令行参数

Usage:
    6.out [-v] [-match pattern] [-benchmarks pattern]

The -v flag causes the tests to be logged as they run.  The -matc
flag causes only those tests whose names match the regular expres
pattern to be run. By default all tests are run silently.  If all
the specified test pass, 6.out prints PASS and exits with a 0 exi
code.  If any tests fail, it prints FAIL and exits with a non-zer
code.  The -benchmarks flag is analogous to the -match flag, but
applies to benchmarks.  No benchmarks run by default.

选项-v执行并记录全部执行的测试。选项-match只执行匹配的测试。
所有测试默认没有输出。如果全部测试通过，打印PASS，返回0后退出。
如果有测试识别，打印FAIL，返回非零值退出。选项-benchmarks启动基准测试。
默认基准测试没有启动。
```

10.1.7. Goyacc

```
Goyacc is a version of yacc for Go.
It is written in Go and generates parsers written in Go.

It is largely transliterated from the Inferno version written in
which in turn was largely transliterated from the Plan 9 version
written in C and documented at

    http://plan9.bell-labs.com/magic/man2html/1/yacc

Yacc adepts will have no trouble adapting to this form of the too

The file units.y in this directory is a yacc grammar for a versio
the Unix tool units, also written in Go and largely transliterate
from the Plan 9 C version.

The generated parser is reentrant. Parse expects to be given an
argument that conforms to the following interface:

    type yyLexer interface {
        Lex(lval *yySymType) int
        Error(e string)
    }

Lex should return the token identifier, and place other token
information in lval (which replaces the usual yylval).
Error is equivalent to yyerror in the original yacc.

Code inside the parser may refer to the variable yylex
which holds the yyLexer passed to Parse.
```

10.1.8. gopack

```
Gopack is a variant of the Plan 9 ar tool.  The original is docur

    http://plan9.bell-labs.com/magic/man2html/1/ar

It adds a special Go-specific section __.PKGDEF that collects al
Go type information from the files in the archive; that section i
used by the compiler when importing the package during compilatio

Usage: gopack [uvnbailo][mrxtdpq] archive files ...

The new option 'g' causes gopack to maintain the __.PKGDEF sectio
as files are added to the archive.
```

10.1.9. gofmt

```
gofmt 程序格式化.

Without an explicit path, it processes the standard input. Given
it operates on that file; given a directory, it operates on all
recursively. (Files starting with a period are ignored.)

没有指定路径，输出到终端。指定了文件，就操作当前文件。
指定了路径就递归指定路径下面的所有.go文件。(Files starting with a perio

Usage:

gofmt [flags] [path ...]

选项:

-l
    just list files whose formatting differs from gofmt's; genera
unless -w is also set.
    只列出gofmt需要格式化的文件，不对文件做任何操作，除非使用 -w
-r rule
    apply the rewrite rule to the source before reformatting.
    在代码格式化之前执行替换规则
-s
    try to simplify code (after applying the rewrite rule, if any
    简化代码(在执行替换或其他的操作后)
-w
    if set, overwrite each input file with its output.
    如果有-w，把格式化后的代码写入原始文件中
-spaces
    align with spaces instead of tabs.
    使用空格替换tab制表符
-tabindent
    indent with tabs independent of -spaces.
    使用tab制表符替换空格
-tabwidth=8
    tab width in spaces.
    tab的长度

调试选项:

-trace
    print parse trace.
    打印跟踪分析
-ast
    print AST (before rewrites).
    打印AST (在重写之前)
-comments=true
    print comments; if false, all comments are elided from the ou
    打印注释，如果是假(false)，所有的注释信息不做处理
```

```
The rewrite rule specified with the -r flag must be a string of t
选项 -r 的重写规则必须遵循这个模式:

pattern -> replacement

Both pattern and replacement must be valid Go expressions. In the
single-character lowercase identifiers serve as wildcards matchin
those expressions will be substituted for the same identifiers in
pattern和replacement必须是有效的Go语法。Pattern 单字符小写标识符作为通配
这些表达式将被替换为相同的标识符.

实例

To check files for unnecessary parentheses:
检查并输出有多余括号的文件

gofmt -r '(a) -> a' -l *.go

To remove the parentheses:
去掉多余的括号

gofmt -r '(a) -> a' -w *.go

To convert the package tree from explicit slice upper bounds to i
slice使用隐形(implicit)替换

gofmt -r 'α[β:len(α)] -> α[β:]' -w $GOROOT/src/pkg

Bugs

The implementation of -r is a bit slow.
选项 -r 的实现方式效率有些低
```

10.1.10. goinstall

```
Goinstall is an experiment in automatic package installation.
It installs packages, possibly downloading them from the internet
It maintains a list of public Go packages at http://godashboard.a

Usage:
    goinstall [flags] importpath...
    goinstall [flags] -a

Flags and default settings:
        -a=false          install all previously installed packag
    -dashboard=true   tally public packages on godashboard.appspo
    -log=true         log installed packages to $GOROOT/goinstall
    -u=false          update already-downloaded packages
    -v=false          verbose operation
```

```
Goinstall installs each of the packages identified on the command
installs a package's prerequisites before trying to install the p
itself. Unless -log=false is specified, goinstall logs the import
installed package to $GOROOT/goinstall.log for use by goinstall -

If the -a flag is given, goinstall reinstalls all previously inst
packages, reading the list from $GOROOT/goinstall.log.  After upd
new Go release, which deletes all package binaries, running

    goinstall -a

will recompile and reinstall goinstalled packages.

Another common idiom is to use

    goinstall -a -u

to update, recompile, and reinstall all goinstalled packages.

The source code for a package with import path foo/bar is expecte
to be in the directory $GOROOT/src/pkg/foo/bar/.  If the import
path refers to a code hosting site, goinstall will download the c
if necessary.  The recognized code hosting sites are:

    BitBucket (Mercurial)

        import "bitbucket.org/user/project"
        import "bitbucket.org/user/project/sub/directory"

    GitHub (Git)

        import "github.com/user/project.git"
        import "github.com/user/project.git/sub/directory"

    Google Code Project Hosting (Mercurial, Subversion)

        import "project.googlecode.com/hg"
        import "project.googlecode.com/hg/sub/directory"

        import "project.googlecode.com/svn/trunk"
        import "project.googlecode.com/svn/trunk/sub/directory"

    Launchpad

        import "launchpad.net/project"
        import "launchpad.net/project/series"
        import "launchpad.net/project/series/sub/directory"

        import "launchpad.net/~user/project/branch"
        import "launchpad.net/~user/project/branch/sub/directory"

If the destination directory (e.g., $GOROOT/src/pkg/bitbucket.org
already exists and contains an appropriate checkout, goinstall wi
```

```
    attempt to fetch updates.  The -u flag changes this behavior,
    causing goinstall to update all remote packages encountered durir
    the installation.

    When downloading or updating, goinstall first looks for a tag or
    named "release".  If there is one, it uses that version of the cc
    Otherwise it uses the default version selected by the version cor
    system, typically HEAD for git, tip for Mercurial.

    After a successful download and installation of a publicly access
    remote package, goinstall reports the installation to godashboarc
    which increments a count associated with the package and the time
    of its most recent installation.  This mechanism powers the packa
    at http://godashboard.appspot.com/package, allowing Go programmer
    to learn about popular packages that might be worth looking at.
    The -dashboard=false flag disables this reporting.

    By default, goinstall prints output only when it encounters an er
    The -v flag causes goinstall to print information about packages
    being considered and installed.

    Goinstall does not attempt to be a replacement for make.
    Instead, it invokes "make install" after locating the package sou
    For local packages without a Makefile and all remote packages,
    goinstall creates and uses a temporary Makefile constructed from
    the import path and the list of Go files in the package.
```

2011-03-02

10.2. 视频和讲座

10.2.1. Go Programming

A presentation delivered by Rob Pike and Russ Cox at Google I/O 2010. It illustrates how programming in Go differs from other languages through a set of examples demonstrating features particular to Go. These include concurrency, embedded types, methods on any type, and program construction using interfaces.

官方： http://www.youtube.com/watch?v=jgVhBThJdXc

优酷： http://v.youku.com/v_show/id_XMTkzOTM4OTA4.html

10.2.2. The Go Tech Talk

An hour-long talk delivered by Rob Pike at Google in October 2009. The language's first public introduction. (See the slides in PDF format.) The language has changed since it was made, but it's still a good introduction.

官方： http://www.youtube.com/watch?v=rKnDgT73v8s

优酷： http://v.youku.com/v_show/id_XMTMxMzIwMTQ4.html]

10.2.3. gocoding YouTube Channel

A YouTube channel that includes screencasts and other Go-related videos:

- Screencast: Writing Go Packages - writing, building, and distributing Go packages.
- Screencast: Testing Go Packages - writing unit tests and benchmarking Go packages.

官方： http://www.youtube.com/gocoding

10.2.4. The Expressiveness Of Go

A discussion of the qualities that make Go an expressive and comprehensible language. The talk was presented by Rob Pike at JAOO 2010. The recording of the event was lost due to a hardware error.

官方： http://golang.org/doc/ExpressivenessOfGo.pdf

10.2.5. Another Go at Language Design

A tour, with some background, of the major features of Go, intended for an audience new to the language. The talk was presented at OSCON 2010. See the presentation slides.

This talk was also delivered at Sydney University in September 2010. A video of the lecture is available here.

官方： http://www.oscon.com/oscon2010/public/schedule/detail/14760

10.2.6. Go Emerging Languages Conference Talk

Rob Pike's Emerging Languages Conference presentation delivered in July 2010. See the presentation slides. Abstract:

Go’s approach to concurrency differs from that of many languages, even those (such as Erlang) that make concurrency central, yet it has deep roots. The path from Hoare’s 1978 paper to Go provides insight into how and why Go works as it does.

官方： http://www.oscon.com/oscon2010/public/schedule/detail/15464

10.2.7. The Go Promo Video

A short promotional video featuring Russ Cox demonstrating Go's fast compiler.

官方： http://www.youtube.com/watch?v=wwoWei-GAPo

优酷： http://v.youku.com/v_show/id_XMTc5MTk3NTY0.html

10.2.8. The Go Programming Language

Go is a new, experimental, concurrent, garbage-collected programming language developed at Google over the last two years and open sourced in November 2009. It aims to combine the speed and safety of a static language like C or Java with the flexibility and agility of a dynamic language like Python or JavaScript. It is intended to serve as a convenient, lightweight, fast language, especially for writing concurrent systems such as Web servers and distributed systems.

This talk will introduce Go's unique feature set, and discuss some of the ways in which it is being used today.

Andrew Gerrand

Developer Advocate

Google Sydney

Andrew Gerrand is a Developer Advocate at Google Sydney where he works on the Go Programming Language. He has given presentations and tutorials on Go in ten countries across three continents. Before joining Google, he spent 10 years

programming for ISPs, web start-ups, and freelance clients in Melbourne and Sydney. In his spare time he writes code for 30-year-old, 8-bit computers.

讲义: http://wh3rd.net/practical-go/

官方： http://osdc.blip.tv/file/4432146

优酷： http://v.youku.com/v_show/id_XMjI1NzQyMjAw.html

10.2.9. Go语言：互联网时代的C

国内的讲座。

优酷： http://v.youku.com/v_show/id_XMTY4Mzk5NTc2.html

2011-03-02

10.3. Release History

This page summarizes the changes between tagged releases of Go. For full details, see the Mercurial change log.

10.3.1. 2010-11-23

```
This release includes a backwards-incompatible package change to
sort.Search function (introduced in the last release).
See the change for details and examples of how you might change y
        http://code.google.com/p/go/source/detail?r=102866c369

* build: automatically #define _64BIT in 6c.
* cgo: print required space after parameter name in wrapper funct
* crypto/cipher: new package to replace crypto/block (thanks Adam
* crypto/elliptic: new package, implements elliptic curves over p
* crypto/x509: policy OID support and fixes (thanks Adam Langley)
* doc: add link to codewalks,
        fix recover() documentation (thanks Anschel Schaffer-Cohe
        explain how to write Makefiles for commands.
* exec: enable more tests on windows (thanks Alex Brainman).
* gc: adjustable hash code in typecheck of composite literals
        (thanks to vskrap, Andrey Mirtchovski, and Eoghan Sherry)
* gc: better error message for bad type in channel send (thanks A
* godoc: bug fix in relativePath,
        compute search index for all file systems under godoc's c
        use correct time stamp to indicate accuracy of search res
* index/suffixarray: use sort.Search.
* net: add ReadFrom and WriteTo windows version (thanks Wei Guang
* reflect: remove unnecessary casts in Get methods.
* rpc: add RegisterName to allow override of default type name.
* runtime: free memory allocated by windows CommandLineToArgv (th
* sort: simplify Search (thanks Roger Peppe).
* strings: add LastIndexAny (thanks Benny Siegert).
```

10.4. Go Roadmap

This page lists features and ideas being developed or discussed by the Go team. This list will be updated as work continues.

The roadmap should be discussed on the golang-nuts mailing list.

10.4.1. Language roadmap

This is a list of language changes that are being considered. Appearance on this list is no guarantee that the change will be accepted.

- Possibly rewrite restriction on goto across variable declarations.
- Variant types. A way to define a type as being the union of some set of types.
- Generics. An active topic of discussion.
- Methods for operators, to allow a type to use arithmetic notation for expressions.

10.4.2. Implementation roadmap

- Improved garbage collector, most likely a reference counting collector with a cycle detector running in a separate core.
- Debugger.
- App Engine support.
- Improved CGO including some mechanism for calling back from C to Go.
- Improved implementation documentation.

10.4.3. Gc compiler roadmap

- Implement goto restrictions.
- Generate DWARF debug info.
- Provide gdb support for runtime facilities.
- Improved optimization.
- 5g: Better floating point support.

10.4.4. Gccgo compiler roadmap

- Implement goto restrictions.
- Use goroutines rather than threads.
- Separate gcc interface from frontend proper.
- Use escape analysis to keep more data on stack.

10.4.5. Done

- Safe compilation mode: generate code that is guaranteed not to obtain an invalid memory address other than via import "unsafe".
- Gccgo: garbage collection.
- Native Client (NaCl) support.
- SWIG support.
- Simpler semicolon rules.
- A more general definition of ... in parameter lists.
- Explicit conversions from string to []byte and []int.
- A function that will be run by the garbage collector when an item is freed (runtime.SetFinalizer).
- Public continuous build and benchmark infrastructure (gobuilder).
- Package manager (goinstall).
- A means of recovering from a panic (recover).

10.5. 相关资源

The Go Blog The Go project's official blog, maintained by the core Go developers.

research!rsc Posts labelled 'Go' by Russ Cox, one of the core Go developers.

Airs Posts labelled 'Programming' by Ian Lance Taylor, one of the core Go developers.

nf.id.au Posts labelled 'Go' by Andrew Gerrand, one of the core Go developers

jmcneil.net Site Perceptive Archive for the 'Go' Category by Jeff McNeil

mprescient.com Entries in golang by Charles Thompson

rogpeppe.wordpress.com

Go Programming Resources

Golang@Reddit

Golang@StackOverflow

Learning Go

Network programming with Go